AFRL-IF-RS-TR-2001-101
Final Technical Report
May 2001

HETEROGENEOUS DISTRIBUTED INFORMATION MANAGEMENT FOR THE INFOSPHERE (HDIMI)

Honeywell Technology Center

Jim Richardson, Mukul Agrawal, N. V. Vaidyanathan, Jaideep Srivastava, Raja Harinath, Wonjun Lee, and Difu Su

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

20010706 107

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-101 has been reviewed and is approved for publication.

APPROVED: Mark D. Foresti

MARK D. FORESTI
Project Engineer

FOR THE DIRECTOR: James A. Collins

JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** May 2001	**3. REPORT TYPE AND DATES COVERED** Final May 96 - Nov 99

4. TITLE AND SUBTITLE
HETEROGENEOUS DISTRIBUTED INFORMATION MANAGEMENT FOR THE INFOSPHERE (HDIMI)

5. FUNDING NUMBERS
C - F30602-96-C-0130
PE - 63728F
PR - 2530
TA - 02
WU - 02

6. AUTHOR(S)
Jim Richardson, Mukul Agrawal, N. V. Vaidyanathan, Jaideep Srivastava, Raja Harinath, Wonjun Lee, and Difu Su

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Prime: Honeywell Technology Center
3660 Technology Drive
Minneapolis Minnesota 55418

Sub: University of Minnesota
Computer Science Department
200 Union Street S.E.
Minneapolis Minnesota 55455

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFTD
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2001-101

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Mark D. Foresti/IFTD/(315) 330-2233

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
The HDIMI project is developing broadly applicable technology for information management in monitoring and control applications such as C4I and industrial control. The key technology is Active Views, an integrated set of mechanisms that provide uniform treatment of state, state change, and state history for arbitrary object types throughout a system. Active Views allow the construction of use-specific "windows on the world" that provide the user with the information he/she needs, extracted from the vast pools of information available on the global infosphere. The source information can take such diverse forms as streaming video and conventional database structures, and can be filtered through a view function before presentation to the user of client application. The view function is defined as a composition of more basic functions drawn from an extensible library. When the source information changes, the corresponding change to the view (if any) is computed and delivered to the user or client application with specified quality of service. The resulting system, called Sonata, includes a CORBA-based distributed run-time environment, a COTS OODBMS for storing structured data, a continuous media server, and a graphical program development tool.

14. SUBJECT TERMS
Object-Oriented Systems, Active Databases, Multimedia, Quality of Services, Application Development Tools, Situation Awareness

15. NUMBER OF PAGES
136

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Section 1 Introduction 1
- 1.1 Project Background and Objectives 2
- 1.2 Technical Approach 3
- 1.3 Accomplishments 4
  - 1.3.1 Active View Management 4
  - 1.3.2 Block-Based Programming Infrastructure 5
  - 1.3.3 Application Development Tools 5
  - 1.3.4 Multimedia Object Management 6
  - 1.3.5 Distributed Resource Management 6
  - 1.3.6 Demonstration Application 7/8
  - 1.3.7 Other Accomplishments 7/8
- 1.4 Report Organization 7/8

Section 2 Active Views 9
- 2.1 Introduction 9
- 2.2 Model of Objects and Types 11
  - 2.2.1 Types 12
  - 2.2.2 Example: Stack 13
  - 2.2.3 Behavior for Sequences of Requests 13
  - 2.2.4 Relationship to Formal Specification Languages 14
  - 2.2.5 Object Equality 15
  - 2.2.6 Compression Functions 16
  - 2.2.7 Distance Functions 17
  - 2.2.8 Examples: Set, Scalar, GeographicLocation 18
  - 2.2.9 Object Containment, Identifiers, and Creation 20
  - 2.2.10 Example: Record 21
  - 2.2.11 Example: SetContainer 22
  - 2.2.12 Objects, Object Identifiers and References, Object Containment, and Values 24
  - 2.2.13 Generating Object Identifiers 25
- 2.3 Views 26
  - 2.3.1 View Specifications 28
  - 2.3.2 Example: SubsetContainer 29
  - 2.3.3 View Implementations and Object Implementations 32
  - 2.3.4 Indirect View Implementations and Inheritance 33
- 2.4 Histories 35
  - 2.4.1 History Type Definition 36
  - 2.4.2 An Object as a View of its History 37
  - 2.4.3 Views of Histories and Information Quality of Service 38
  - 2.4.4 Histories of Views 41
- 2.5 Object Synchronization 44
  - 2.5.1 SynchronizationControl Type Definition 46
  - 2.5.2 Synchronizer View Specification 48
  - 2.5.3 Synchronizer Procedure Implementation 49

2.5.4 Discussion 50
2.6 Relationship to Active Databases 51
2.7 Future Work 53
Section 3 Block-Based Programming 55
3.1 Presto Programming Model 55
3.2 Extensions to Support Active Views 57
3.3 Re-Implementation of the Distributed Execution Environment 58
3.4 Conclusion 60
Section 4 Application Development Tools 61
4.1 Introduction 61
4.1.1 PDT Overview 61
4.1.2 Integrated Toolkit Environment 62
4.1.3 Programming Model 62
4.2 The Need for PDT 64
4.2.1 Requirement Specification 64
4.2.2 Input-Output Specification 65
4.2.3 Integration with UIDT 65
4.3 Design Issues 65
4.3.1 Block and Port Design 66
4.3.2 Connection Design 67
4.4 Implementation Issues 68
4.4.1 Language Issues 68
4.4.2 User Interface Issues 69
4.4.3 Integrated Toolkit Issues 69
4.5 Proposal for Further Research 71
4.5.1 Interaction with CMT 71
4.5.2 Implementation of PAT 72
4.6 Conclusion 72
Section 5 Continuous Media Server 73
5.1 Introduction 73
5.2 Related Works 74
5.3 Motivation & Objectives 75
5.3.1 Resource Constraints 75
5.3.2 QoS Metrics 75
5.3.3 Objectives 76
5.4 Design of CM Server 77
5.4.1 Overview of Architecture 77
5.4.2 CM Client Architecture 78
5.4.3 Network Manager 78
5.4.4 Proxy Server 79
5.4.5 CORBA Implementations 81
5.4.6 QoS Manager 81
5.4.7 I/O Manager 83
5.5 Implementation Issues 83
5.5.1 Server Model 83
5.5.2 Number of Threads 84

5.5.3 File System 84
5.5.4 Packet Size 84
5.5.5 Integration with CORBA 84
5.5.6 Other Issues 85
5.6 Conclusions 85
Section 6 Demonstration Application: Planning and Monitoring Air Combat 87
6.1 Application Overview 88
6.2 User Interfaces 89
6.3 Schema 89
6.4 Transactions 93
6.4.1 AirPlatformType 93
6.4.2 TargetStrategy 94
6.4.3 TargetObjective 94
6.4.4 AirPlatform 95
6.5 Scenario 95
6.5.1 Database Initialization 96
6.5.2 Run-Time Scenario 97
Section 7 Conclusions 99
7.1 Dual-Use Applications 99
7.2 Recommendations for Future Work 99
7.2.1 Active Collaboration Information Server (ACIS) 100
7.2.2 ACIS-Application Integrated Environment (ACIS-AIE) 100
7.2.3 Quality-Adaptive Information Management (QAIM) 101
7.2.4 Temporal ACIS 102
Section 8 References 105
Section 9 Publications and Patents 109
9.1 Publications 109
9.2 Patents 111/112
Section 10 Software Documentation 113
10.1 Application Development Tools 113
10.1.1 Block Class 113
10.1.2 Connection Class 115
10.1.3 ProgFile Class 115
10.1.4 SonataFile Class 117
10.1.5 SonataGUI Class 118
10.1.6 SonataCanvas Class 118
10.1.7 Utility Classes 119
10.1.8 Conclusion 120
10.2 Continuous Media Server 120
10.2.1 Introduction 120
10.2.2 Program Modules 121
10.2.3 Functions in Source Modules 122

List of Figures

Figure 1. Warriors need relevant and timely information from the global Infosphere 1

Figure 2. Sonata Reference Architecture 3

Figure 3. View 10

Figure 4. History 10

Figure 5. Object Synchronization 10

Figure 6. An Application of Active Views 11

Figure 7. Constructing a Containment Relationship Using a View 25

Figure 8. Graphical Representation of a View Specification and a View Implementation 27

Figure 9. View Implementation Operation 32

Figure 10. Indirect Implementation of SizeOf View 34

Figure 11. A Wrapper Implementation of a History 38

Figure 12. An Implementation of the IQoS View Specification 40

Figure 13. The View Relationships Among a View Object, Its Source Objects, and Their Histories 41

Figure 14. An Implementation of a History of a View 43

Figure 15. Object Synchronization Objects, View Specification, and View Implementation 45

Figure 16. Example of Object Synchronization 46

Figure 17. Example Block-Based Program 55

Figure 18. Example Program Blocks 56

Figure 19. Port Types 56

Figure 20. User Program to System Program Translation 57

Figure 21. Internal Structure of a Block 58

Figure 22. Interactive Video-on-Demand 63

Figure 23. A Composite Block 66

Figure 24. Zoom In View 67

Figure 25. Two Example Streams Used to Explain QoS Metrics 77

Figure 26. Architecture of Continuous Media Server 78

Figure 27. Different applications require different views of plan and situation data. 87

Figure 28. PMAC Application Schema 89

Section 1
Introduction

The objectives of the Heterogeneous Distributed Information Management for the Infosphere (HDIMI) project have their origin in *C^4I for the Warrior* [J6I93]. That document stated the importance of providing individual warriors, whatever their role, with relevant and timely information from the global Infosphere (see Figure 1). More recently, DoD's *Joint Vision 2010* [CJCS98] has emphasized the key role of information superiority in achieving military success. Information superiority leads to enhanced *battlespace awareness* (understanding of the current military situation) and *speed of command* (ability to plan and execute operations to meet objectives and to adapt to changing situations.)

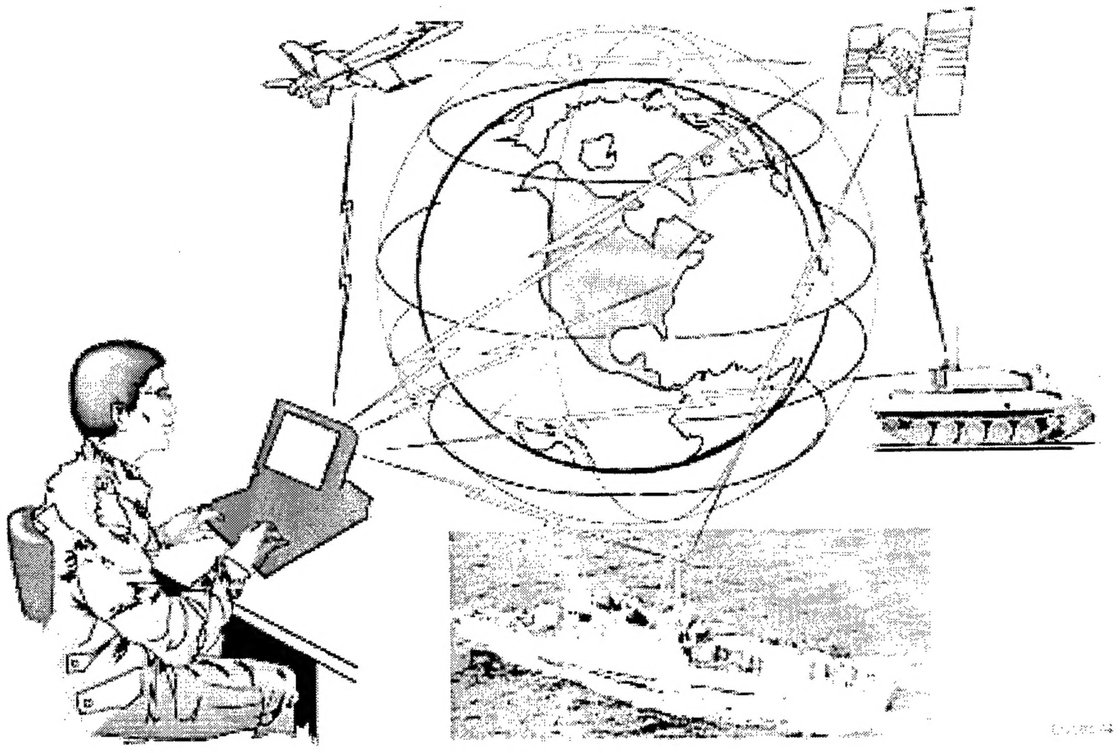


Figure 1. Warriors need relevant and timely information from the global Infosphere.

The HDIMI project is conducting research and development in information management technology to support these operational objectives. This introduction covers:

- HDIMI project background and objectives,
- The technical approach as originally proposed,
- Actual project accomplishments, and
- The organization of the remainder of this report.

Honeywell Technology Center and the University of Minnesota have conducted the HDIMI project under Rome Laboratory (now Air Force Research Laboratory) sponsorship.

1.1 Project Background and Objectives

The HDIMI project is a successor to the Multimedia Database Management System project conducted in 1993–1996 under Rome Laboratory sponsorship. That project focused on system services and tools to support continuous media (audio, video) in time-critical C^4I applications. Significant accomplishments in that project included:

- A block-based programming model and graphical tool for dynamic construction of complete continuous media applications.
- A multi-resource run-time scheduling component that ensured continued execution of critical applications when system resources are tight, while allowing others to operate with reduced quality of service.
- A high-performance multimedia file system.

All of these capabilities were implemented in a Solaris-based system called P*resto*. P*resto* was subsequently extended to a distributed environment under a DARPA-sponsored project, High Performance Network Services [AKPBV96].

The HDIMI project has taken many of the concepts embodied in P*resto* and generalized them to address information management requirements implicit in *C^4I for the Warrior* and *Joint Vision 2010*. Specifically, the HDIMI project objectives are to *"investigate, develop, and demonstrate techniques for meeting C^4I system application requirements:*

- *A wide variety of information, including conventional data types and continuous media, stored in a collection of heterogeneous data sources in a distributed environment*
- *A means for each application to defines its 'window on the world' and to specify policies on how closely the window must be kept in synch with the global Infosphere*
- *The operation and coexistence of QoS-sensitive C^4I applications and other C^4I applications*
- *Quick and easy prototyping of C^4I applications*

...within the framework of an overall layered system architecture."

Significant requirements beyond the capabilities of P*resto* include:

- Support for data types that lack a time dimension, e.g. text, images, and conventional database structures. (P*resto* only supported continuous media.)
- Ability to define a "window on the world", i.e. an application- or user-specific *Active View* of the global Infosphere. (P*resto* supported development of stand-alone continuous media applications. It did not support multiple concurrent accesses to shared databases.)
- More general notions of Quality of Service for these views. (P*resto* supported QoS measures specific to continuous media.)

- Distributed multi-resource management. (P*resto* supported CPU and memory resource management on a single node only.)

1.2 Technical Approach

The evolution of the P*resto* system to accommodate the new requirements is called *Sonata*. The Sonata reference architecture (Figure 2) has evolved from P*resto* to align with the Joint Task Force Architecture Specification (JTFAS) being developed under DARPA sponsorship [TFS94]. The functional components comprising the architecture fall into five categories:

- *COTS components* are commercially available hardware and software products.
- P*resto components* were built during the Multimedia Database Management System project and require no significant revision.
- *Revised* P*resto components* are those that were initially developed in the Multimedia Database Management System project and need substantial extensions in the HDIMI project.
- *New Sonata components* are being developed from scratch under the HDIMI project.
- *Future work*, such as C^4I applications, can be developed by a follow-on project after successful execution of the proposed HDIMI project.

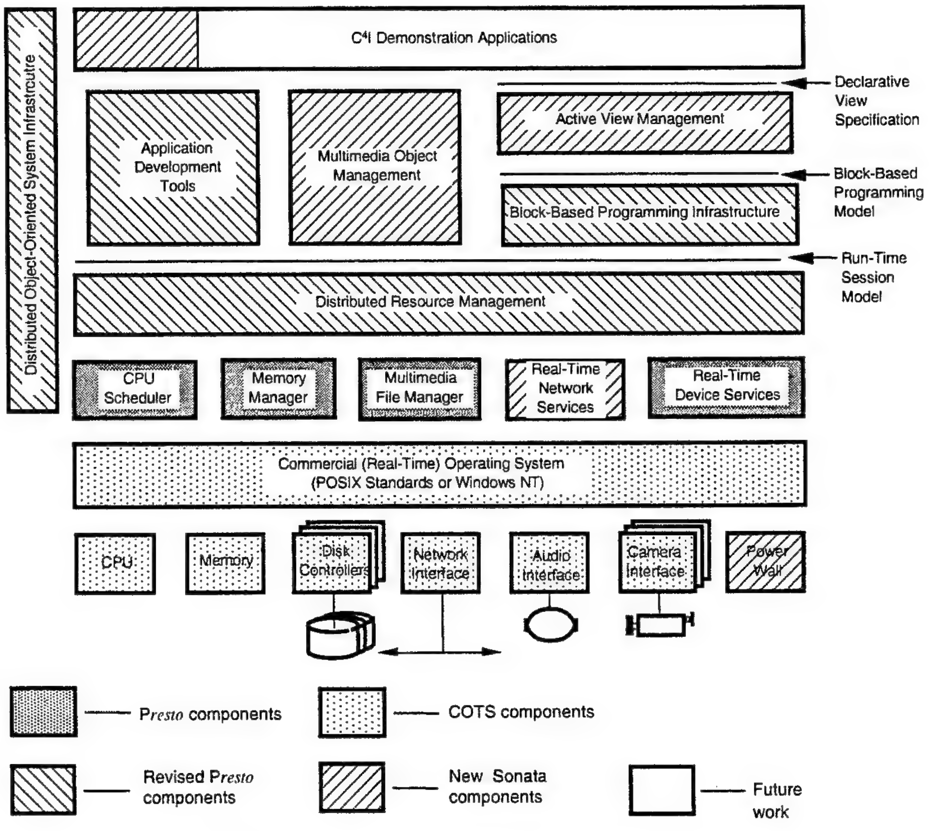

Distributed Object-Oriented System Infrastructure
C⁴I Demonstration Applications
Declarative View Specification
Application Development Tools
Multimedia Object Management
Active View Management
Block-Based Programming Model
Block-Based Programming Infrastructure
Run-Time Session Model
Distributed Resource Management
CPU Scheduler
Memory Manager
Multimedia File Manager
Real-Time Network Services
Real-Time Device Services
Commercial (Real-Time) Operating System (POSIX Standards or Windows NT)
CPU
Memory
Disk Controllers
Network Interface
Audio Interface
Camera Interface
Power Wall
Presto components
COTS components
Revised Presto components
New Sonata components
Future work

Figure 2. Sonata Reference Architecture

At the outset of the HDIMI project, we proposed to develop the following six major capabilities. Actual accomplishments are listed in the next subsection.

- *Active view management*—Develop Active View capabilities for the multimedia infosphere. This component consists of a declarative view specification model and language, and algorithms that map the views specified in the language to a data-flow-oriented, function-block-based program for execution.
- *Block-based programming infrastructure*—Supporting the Active View management, this component consists of a block-based programming interface and view execution mechanisms. The block-based programming model (and an associated program development tool) was designed and prototyped in the Multimedia Database Management System project. Enhance this model with the capabilities of operation flow support and distributed, location-transparent view definition and execution.
- *Application development tools*—Extend the visual program development tool prototyped in the Multimedia Database Management System project to support Active View specification in addition to application construction. Develop a user interface tool to facilitate construction of user interfaces for different applications, and a program analysis tool.
- *Multimedia object management*—Develop capabilities of heterogeneous data type management and content-based query for the multimedia infosphere. Prototype this component using a commercial database management system.
- *Distributed resource management*—This is an extension of the P*resto* work. Investigate and develop distributed scheduling techniques on top of POSIX-compliant commercial operating systems to support the Active View capability.
- *Application Demonstration*—This indicates various demonstration application software components to be built in the system environment. Develop a set of software components in the context of a DoD demonstration to illustrate the capabilities of all the system components described above.

1.3 Accomplishments

We substantially accomplished the HDIMI project objectives, as summarized below. Subsequent sections of this report provide further details.

1.3.1 Active View Management

We defined Active View services, which involve three major concepts: view, history, and ojbect synchronization. We have implemented the view and object synchronization concepts in multiple distributed demonstration applications. The service definitions have proved remarkably robust over time.

Section 2 describes Active Views more fully.

1.3.2 Block-Based Programming Infrastructure

P*resto* used a data-flow oriented, block-based programming model and execution environment for continuous media applications. The Multimedia Database Management System project implemented this environment for a single Solaris node. Subsequently, Honeywell Technology Center used and extended the environment under a DARPA-funded project, High Performance Network Services. This distributed environment was the starting point for the HDIMI project.

We developed extensions to P*resto*'s data-flow oriented, block-based programming model to support operation flows in addition to data flow, and to handle aperiodic flows in addition to periodic flows. These changes were necessary to implement Active View services, and moved the range of applications well beyond the continuous media applications that P*resto* supported.

We replaced P*resto*'s custom-built distributed execution environment with one based on CORBA. Specifically, we use Iona's Orbix product. While this involved replacing major portions of the Sonata code, we believe it provided the best chance of transferring the technology to DARPA programs.

We developed a library of reusable view functions that are building blocks for new applications. These applications can be defined using the application development tools defined below.

The block-based programming model and infrastructure are described further in Section 3.

1.3.3 Application Development Tools

We re-implemented P*resto*'s Program Development Tool (PDT) in Java. The previous version depended on a Smalltalk-based "meta-tool" called DoME that, while powerful, required too much specialized knowledge to use. The Java implementation will permit relatively easy porting to other platforms in the future.

We developed a User Interface Development Tool (UIDT) to construct application user interfaces in a "visual" manner consistent with Active Views concepts. The UIDT is integrated with the PDT, so that the same application can easily be viewed from either perspective.

We have achieved levels of tool/run-time integration and data type support that are significant advances over P*resto*:

- In P*resto*, the PDT was used to define a complete program, from a continuous media source (e.g. camera or video file), through data transformation functions (blocks) to sink (e.g. file or display). The program was then run as a unit on one or more Sun workstations.
- In Sonata, programs are built incrementally. Data sources include ObjectStore class extents *and* continuous media sources; views can be built on top of them, and on top of previously defined views. The CORBA name service holds the set of data sources and views available at any given time. These are visible in the PDT for further view construction. Applications can be built and executed *incrementally*, adding new views (blocks) on top of views that are already active in the run-time environment. These applications run in a distributed environment, linked via CORBA.

Applications developed using PDT and UIDT can be targeted to multiple execution environments. In addition to targeting applications to the Sonata run-time system, the tools can develop applications for the Berkeley Continuous Media Toolkit run-time [SMITH94, PATEL95].

Section 4 covers these application development tools in more detail.

1.3.4 Multimedia Object Management

We evaluated a number of COTS object-oriented and object-relational database management systems to use as a basis for persistent object management and query. We selected Object Design Inc.'s ObjectStore server.

We developed tools to facilitate creation of Active Views of arbitrary ObjectStore schemas.

We extended P*resto*'s continuous media file system into a Continuous Media Server (CMS) that supports concurrent retrieval of multiple media streams by distributed clients.

CMS is integrated with Active Views—The PDT and UIDT can be used to develop applications that access information from both ObjectStore and CMS.

We demonstrated that P*resto*'s continuous media file system can perform significantly better than the standard UNIX file system.

The Continuous Media Server is described further in Section 5. The COTS DBMS evaluation is reported separately [PS97].

With AFRL concurrence, we decided not to investigate content-based query of multimedia data. The combination of content-based query and Active View technology is a powerful one for automating intelligence data analysis. For instance, one could define a view that lists enemy tanks in a specified geographic region, given a set of raw images. Computing the view requires executing image analysis algorithms. Our approach had been to integrate existing algorithms in the Active View framework, rather than to innovate in image analysis. However, we believe that the current state of the art of image analysis algorithms is insufficient for a compelling demonstration.

1.3.5 Distributed Resource Management

Resource management was a major focus of the Multimedia Database Management System project. P*resto* included a component that performed admission control and adaptive multi-resource scheduling based on applications' Quality of Service (QoS) needs. We had planned to extend this capability to a distributed environment in Sonata. However, in concurrence with AFRL, we decided not to pursue this objective for several reasons:

- Applying project resources to other objectives (e.g. a more substantial demonstration) was more valuable.
- With the conversion to a CORBA-based run-time infrastructure, the resource management architecture would have required a redesign.

- The resource management concepts developed in the Multimedia Database Management System project are being extended under a separate project funded under DARPA's Quorum program [HJHM+97].

1.3.6 Demonstration Application

We developed several demonstrations in the course of the HDIMI project. We developed technology-oriented demonstrations of Active Views, the Continuous Media Server, and the application development tools. All of these technologies have been incorporated into a larger demonstration that shows how Sonata technologies apply to air combat planning and monitoring. Section 6 describes this demonstration. Instructions for operating the demonstration software are documented separately [USER98].

1.3.7 Other Accomplishments

Honeywell Technology Center and the University of Minnesota have published numerous papers and filed several patent applications related to P*resto* and Sonata. Section 9 lists these.

1.4 Report Organization

The remainder of this report is organized as follows. Sections 2 through 6 cover the main Sonata components at the conceptual level. They introduce the main technical concepts, identify research issues and solutions, and compare our work to other research. Our conclusions and recommendations for future work are set forth in Section 7. Section 8 lists documents referenced in the text. Section 9 lists related publications, proposals, and patent applications. Section 10 documents the design of the Sonata software.

Section 2
Active Views

2.1 Introduction

Traditionally, information systems have been transaction-oriented: the system maintains a body of information in a database or other repository, and users query and update that information at their own initiative. The information system is passive—it doesn't "speak unless spoken to". The trend now is toward more active information systems that inform users or take other actions when new and relevant information enters the system. Examples include collaborative systems and monitoring and control systems in a variety of application domains.

Approaches for providing active behavior generally involve some notion of *event*. CORBA [OMG], COM [CHAP96], and Java [FLAN97] all support events, and there is a large literature on active databases. (See [WC96] for a survey.)

Active Views are an integrated set of mechanisms for constructing active information systems. They are based on a simple, but very specific definition of an event: *an event is a complete description of a change of state in a specific, identified object*. The mechanisms—*view*, *history*, and *object synchronization*—provide a uniform treatment of state, state change, and state history for arbitrary object types (databases, user interfaces, sensors, etc.) throughout a system.

A view (Figure 3) is an object whose state is a function of the states of one or more other objects, called source objects. We extend the conventional (database) concept of view in several ways:

- The source objects need not be persistent.
- The source and view objects can be of arbitrary types, not just collections or other system-provided types.
- The functional relationship $O = F(O_1, \ldots, O_n)$ between the view and source objects can be any computable function, not just a function defined by a system-provided query language.

Because a view is an object, a client application can access or (where meaningful) update the view's state using the operations defined by its type. When a source object's state is updated, the view's state changes to maintain the functional relationship. Similarly, updating the view's state causes an update to one or more source object states.

A history (Figure 4) is a record of an object's past and current states, with services for retrieving the changes between pairs of states. It is an application of the database log concept to arbitrary objects. The state changes are defined by the object's type—they are the type-specific operations that cause a change to the object's state. The object can be a view; the history then records view's state changes, not those of the view's source objects.

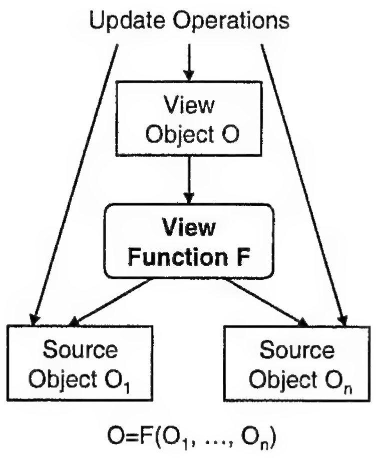


Figure 3. View

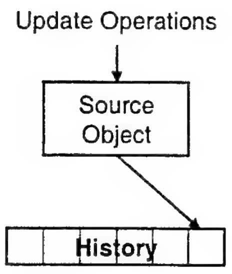


Figure 4. History

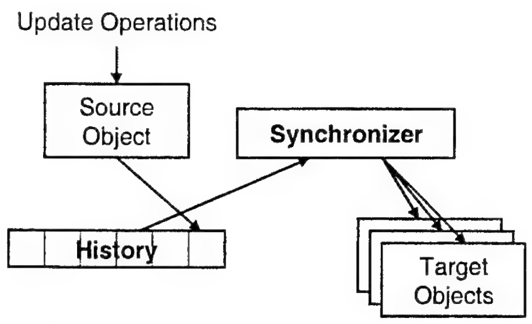


Figure 5. Object Synchronization

Object synchronization (Figure 5) is the propagation of a source object's state changes to one or more target objects. A *synchronizer* monitors the source object's history for new state changes, and propagates them to a set of target objects. The source object can be a view, so that the states of the target objects are synchronized with the view's state, not with the states of the view's source objects.

These three mechanisms support construction of active information systems in the following way:

- An object's history captures its state changes (events) for transmission to interested recipients via object synchronization. This is *information push.*
- Alternatively, a client application can access an object's state via the operations defined by the object's type. This is *information pull.*
- A client application may require a computed transformation of the states or state changes of one or more objects. This can be accomplished by defining a view of those objects, where the view function performs the required transformation.

Figure 6 shows how Active Views could be used to generate an animated map and maintain synchronized copies of the map on multiple displays. The state of the animated map is defined by the following equation:

$$\begin{aligned} AnimatedMap = {} & OverlayIcons(MapCrop(DigitalMap, GPSReceiver), \\ & GenerateIcons(RecordSubset(SituationDB, GPSReceiver))) \end{aligned} \tag{1}$$

The map shows enemy flights over a region of interest centered about a warfighter's current location. There are three source objects: a digital map of the world, a situation database that has current locations for all known enemy aircraft, and a GPS receiver that senses the warfighter's location. The animated map changes whenever the digital map changes in the region of interest (e.g. a bridge has been destroyed); aircraft positions change within the region of interest; or the warfighter moves.

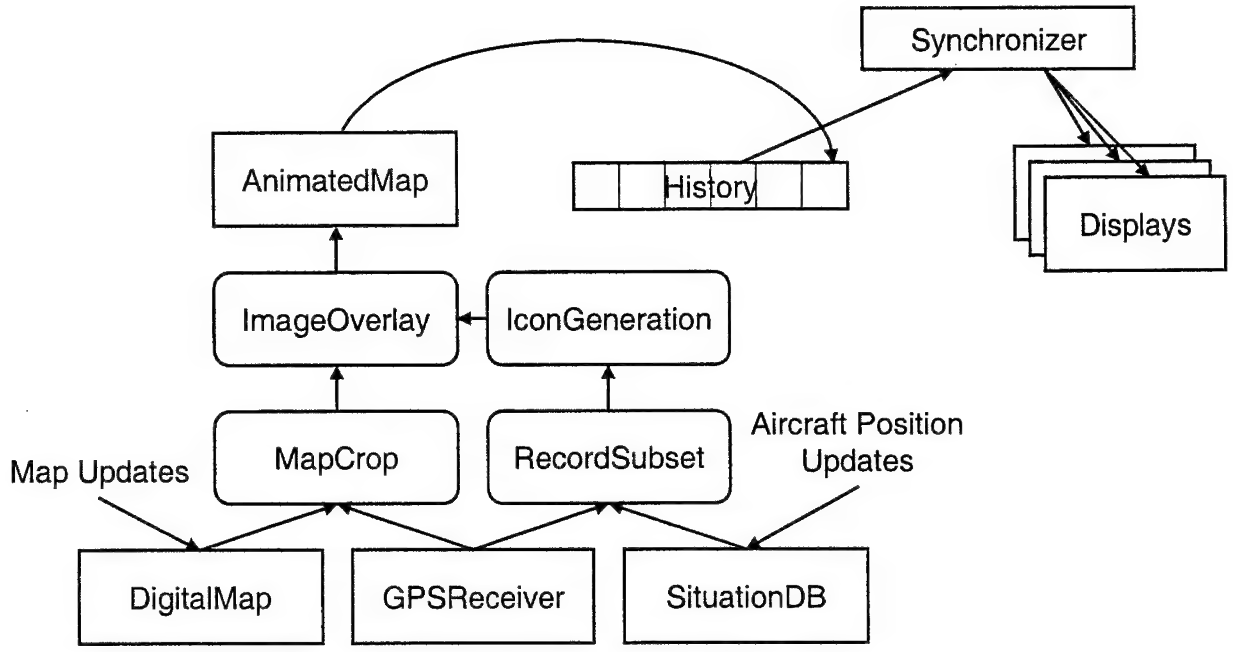


Figure 6. An Application of Active Views

We have implemented Active Views concepts for two widely divergent applications: military air operations planning and video capture, storage and display. Other potential applications include industrial process monitoring and shared collaborative workspaces. Our implementation is based on OMG's CORBA, but the approach could be implemented in other object infrastructures such as Microsoft's OLE/COM or more specialized systems.

Equation (1) is a declarative statement of an information need. It says nothing about the mechanics of updating the map in response to changes in the source objects. The declarative nature of view definitions speeds application development. (For simplicity, the size of the geographic region to be displayed is assumed to be built into the *MapCrop* and *RecordSubset* functions.) Our implementation provides a library of reusable view functions and a graphical tool for composing new functions from library elements. The run-time system takes this view definition and instantiates an efficient, event-driven mechanism for updating the animated map at run-time in the distributed system. The event-oriented nature of object synchronization provides rapid propagation of state changes from source objects, through application-specific view functions, to information consumers.

The remainder of the paper is organized as follows. Section 2.2 presents our model of objects and types, which forms the basis for Active Views. Sections 2.3 through 2.5 define views, histories, and object synchronization more precisely, discuss implementation approaches, and compare these mechanisms individually to similar concepts in the literature. Section 2.6 compares Active Views as an integrated set of mechanisms to active databases as currently defined in the literature. Section 2.7 suggests areas for future work.

2.2 Model of Objects and Types

An *object* is a software entity that interacts with other software entities using a request/response protocol. In this protocol, a client application (which may itself be an object) sends the object a *request*. Each request elicits a *response* from the object and may cause a change in the object's *abstract state* [MEYE97]. A request identifies the operation to be performed on an object and any parameter values for the operation. The response may be a result or an exception indicator. Requests, responses, and abstract states are mathematical values represented in computer memory.

An object also has an *implementation*, which includes an implementation state recorded by one or more other (implementation) objects, and a procedure that receives requests and generates responses based on the request and the current states of the implementation objects. It may also update the states of the implementation objects. There is a close relationship between object implementations and views; this is discussed further in Section 2.3.3.

The remainder of this section documents our models of object types, equality, creation, containment, references, and other common object-oriented concepts that Active Views must deal with in a realistic object-oriented environment. We illustrate the concepts with several examples.

2.2.1 Types

A *type* is an assertion about an object's behavior as viewed by clients. It defines the response that an object of that type will generate as a result of a particular request, and how the object's abstract state will change. For example, a definition of object type *Stack* defines the behavior of an object in response to *push* and *pop* requests. The type in effect defines a contract between an object and the applications that use it. CORBA, COM, and other object-oriented system infrastructures do not address object types or behavior specifications in this sense. Rather, they deal with object *interfaces*, which merely identify the names of permissible operations on an object and the parameters of those operations.

We use the following approach for defining object types. Its relationship to formal specification languages such as Larch [GH93] and Z [SPIV98] will be explained below. An object type T is defined by[1]:

- A *state domain* S_T, *request domain* Q_T, and *response domain* R_T. These are arbitrary mathematical sets.
- A *transition domain* $SQ_T \subseteq S_T \times Q_T$, the set of (state, request) pairs for which the type defines an object's behavior. A type may not define the response of an object for every request in every state; the transition domain effectively defines the precondition for each request.
- A *state transition function* $!_T : SQ_T \to S_T$ and a *response function* $?_T : SQ_T \to R_T$ that define the new state s' and response r for each $(s,q) \in SQ_T$. They are infix functions: $s' = s\,!_T\, q$ and $r = s\,?_T\, q$.

[1] Much of the mathematical notation used here is adopted from Meyer's *Introduction to the Theory of Programming Languages* [MEYE90] This text is very accessible to non-mathematicians (including the authors) but is unfortunately out of print. Similar notation is defined in [SPIV98].

- An *initial state* $\varepsilon_T \in S_T$. We assume that all objects of a given type have a specific, known initial state.

The subscript T will be omitted when the type is understood from context.

This form of type definition is like a finite state transducer [AU72], but here the state, request, and response domains can be infinite.

A type definition can be *parameterized.* A parameterized type is denoted $T[p_1,\ldots,p_n]$. A parameter can be a type, a value domain, or a value drawn from a domain. We will define many parameterized types, including the following example.

2.2.2 *Example: Stack*

The $Stack[X]$ type defines an infinite-capacity stack with elements drawn from value domain X.

$$\begin{aligned}
&S = X^* \\
&Q = \{push(x) \mid x \in X\} \cup \{pop, isEmpty\} \\
&R = \{ok, true, false\} \cup X \\
&SQ = (S \times Q) - \{(\langle\,\rangle, pop)\} \\
&\varepsilon = \langle\,\rangle
\end{aligned}$$

The definition of SQ indicates that the result of a pop request on an empty stack is not defined. The state transition and response functions are defined as follows:

$$\begin{aligned}
&\langle x_1,\ldots,x_n\rangle !\, push(x) = \langle x_1,\ldots,x_n,x\rangle \\
&\langle x_1,\ldots,x_n\rangle ?\, push(x) = ok
\end{aligned}$$

$$\begin{aligned}
&\langle x_1,\ldots,x_n\rangle !\, pop = \langle x_1,\ldots,x_{n-1}\rangle \\
&\langle x_1,\ldots,x_n\rangle ?\, pop = x_n
\end{aligned}$$

$$\begin{aligned}
&\langle x_1,\ldots,x_n\rangle !\, isEmpty = \langle x_1,\ldots,x_n\rangle \\
&\langle x_1,\ldots,x_n\rangle ?\, isEmpty = (n = 0)
\end{aligned}$$

2.2.3 *Behavior for Sequences of Requests*

For any state transition and response functions $!_T$ and $?_T$, there are related functions $!!_T$ and $??_T$ that show the effect of *sequences* of requests on an object. The function $!!_T$ computes the effect on an object's state of sending it a sequence of requests; the function $??_T$ computes the sequence of responses that result from this sequence of requests:

$$\begin{aligned}
&s !!_T \langle\,\rangle = s \\
&s !!_T \langle q_1,\ldots,q_n\rangle = (s !!_T \langle q_1,\ldots,q_{n-1}\rangle) !_T\, q_n
\end{aligned}$$

$$s\,??_T\langle\,\rangle = \langle\,\rangle$$
$$s\,??_T\langle q_1,\ldots,q_n\rangle = \langle s\,?_T\, q_1\rangle \mid (s!_T\, q_1)\,??_T\langle q_2,\ldots,q_n\rangle$$

Here, | denotes sequence concatenation. These functions are defined on a subset SQ_T^* of $S_T \times Q_T^*$, where Q_T^* is the set of sequences of elements of Q_T. SQ_T^* represents the legal sequences of requests that can be sent to an object of type T when the object is in a given state s. It can be defined inductively from SQ_T and $!_T$:

$$SQ_T^0 = S \times \{\langle\,\rangle\}$$
$$SQ_T^n = \{(s,\langle q_1,\ldots,q_n\rangle) \mid (s,q_1) \in SQ_T \wedge (s!_T\, q_1, \langle q_2,\ldots,q_n\rangle) \in SQ_T^{n-1}\} \text{ for } n > 0$$
$$SQ_T^* = \bigcup_{n\geq 0} SQ_T^n$$

The *difference* $s' -_T s$ between two states in S_T is the set of *paths* from s to s', i.e. the set of elements of Q_T^* that change the state of an object of type T from s to s':

(2) $\quad s' -_T s = \{l \mid (s,l) \in SQ_T^* \wedge s!!_T\, l = s'\}$

There may be zero, one, or more than one path from s to s'. State s' is *reachable* from state s (denoted $s \rightarrow_T s'$) if $s' -_T s \neq \emptyset$. We assume that all elements of S_T are reachable from ε_T.

It is easy to show that for any three states s, s', and s'',

(3) $\quad (s'' -_T s') \,\|\, (s' -_T s) \subseteq (s'' -_T s)$

Here, || denotes pairwise concatenation of the elements of its arguments. In other words, the paths from s to s'' include those that pass through s'. It is also easy to see that $\rightarrow_T$ is transitive and reflexive.

2.2.4 Relationship to Formal Specification Languages

The notation used here to define object types is not intended to replace formal specification languages such as Larch or Z. Those languages were designed with expressiveness and formal verification in mind. This notation differs from those languages in at least two ways. First, it models requests as *values*. Z, Larch, and similar specification languages model requests as applications of functions or invocations of procedures or methods. Modeling requests as values allows us to reason more easily about sequences of requests, as above, and types that can deal with any form of request, such as a history. On the other hand, the notation is somewhat more awkward for defining specific types.

Second, our notation can produce only *complete* type specifications—specifications that completely determine the behavior of an object of that type [GH93,MEYE97]. A type's state transition and response functions define the result of each request completely as a specific new state and response. If $\langle q_1,\ldots,q_n\rangle$ is the sequence of requests sent to an object of type T since its creation, and if T defines the behavior for this sequence, i.e. if $(\varepsilon_T, \langle q_1,\ldots,q_n\rangle) \in SQ_T^*$, then the object's state must be $\varepsilon_T \,!!_T \langle q_1,\ldots,q_n\rangle$.

It is not possible to define constraints on the new state or the response while not specifying them completely. For instance, one might want to define a *Set* type with a *selectOne* operation that returns an arbitrary element of a set object and removes that element from the set. As another example, one might want to define a *stack* type with a fixed capacity beyond which *push* requests fail, but leave the actual capacity unspecified. Larch and Z can specify such types, but our notation can't.

This restriction is actually necessary for our purposes. The definitions of view, history, and object synchronization require completely specified types: when a request is applied to an object of type T in a known state, the resulting state *must be completely predictable.*

In practice, such complete predictability can't be achieved. Memory and other resource limitations, hardware faults, and other factors make it impossible to guarantee that an object meets its type definition at all times. To cover these cases, an object must be able to raise an *implementation exception*—a signal that its implementation has failed to meet its type definition. How such implementation exceptions can be gracefully integrated into Active Views mechanisms is a subject of future study.

2.2.5 Object Equality

Object synchronization requires a well-defined notion of object *equality*. But what does it mean for two objects to be equal? The question only makes sense for two objects whose types have the same state domain. In this case, the objects are equal if their states are equal. For synchronization, we impose the further constraint that the two objects be of the same type. This guarantees that requests sent to one of the objects can be sent to the other object, with the same effect on their states.

Object-oriented systems make heavy use of references between objects, and so it is common to define two types of equality, *shallow* and *deep* [MEYE97], which differ in their treatment of references within an object's state. Two objects are "shallowly" equal if their states are equal. If their states include references to other objects, those references must be equal, i.e., they must refer to the same object. Two objects are "deeply" equal if their states are equal and (recursively) the states of any objects they reference are equal.

Our definition of equality—two objects are equal if their states are equal—is shallow equality. What if we want to make a copy of a complex web of interrelated objects, and keep the copy synchronized with the original? This would appear to be impossible with shallow equality, but it isn't. One object can be *contained* by another object, so that the state of the container object includes the state of the contained object. Therefore, a collection of interrelated objects can be copied and synchronized if they are in a container object; the states of the original container object and its copy are kept synchronized according to the shallow equality criterion. See Section 2.2.9 for a discussion of object containment.

2.2.6 *Compression Functions*

The type definition for a history (Section 2.4.1) requires a compression function. A compression function transforms one sequence of requests in $s' -_T s$ to another, generally shorter sequence. More formally, a compression function for type T is a function $c : SQ_T^* \to Q_T^*$. that has the following properties:

$$\forall (s,l) \in SQ_T^* \; s!!_T \, c(s,l) = s!!_T \, l$$
$$\forall (s,l) \in SQ_T^* \; c(s,c(s,l)) = c(s,l)$$

In other words, the compression function generates a sequence of requests that has the same effect on the current state of the object as the original sequence, and applying compression a second time to a sequence of requests has no effect.

Here is an example compression function for type $Stack[T]$. This function is defined recursively, and either reduces the length of the sequence or leaves the sequence unchanged.

$$c(\langle x_1,\ldots,x_n\rangle,\langle q_1,\ldots,push(x),pop,\ldots,q_n\rangle) = c(\langle x_1,\ldots,x_n\rangle,\langle q_1,\ldots,q_n\rangle)$$
$$c(\langle x_1,\ldots,x_n\rangle,\langle q_1,\ldots,isEmpty,\ldots,q_n\rangle) = c(\langle x_1,\ldots,x_n\rangle,\langle q_1,\ldots,q_n\rangle)$$
$$c(\langle x_1,\ldots,x_n\rangle,\langle \underbrace{pop,\ldots,pop}_{m+1 \text{ times}}, push(x_{n-m}),q_1,\ldots,q_n\rangle)$$
$$= c(\langle x_1,\ldots,x_n\rangle,\langle \underbrace{pop,\ldots,pop}_{m \text{ times}},q_1,\ldots,q_n\rangle) \quad 0 \le m < n$$
$$\text{otherwise } c(\langle x_1,\ldots,x_n\rangle,\langle q_1,\ldots q_n\rangle) = \langle q_1,\ldots q_n\rangle$$

Note that the compression function is sensitive to the state of the stack: a *pop* request followed by a $push(x)$ request can be "compressed out" only if x is the stack element that the *pop* request removed. It can be shown that this compression function generates a minimum-length sequence of requests, and that this sequence is unique.

It is possible to define a compression function c that is insensitive to the object's state, i.e. $\forall (s,l) \in SQ_T^* \; c(s,l) = c'(l)$ for some *stateless compression function* c'. A stateless compression function simplifies the implementation of a history. It can be defined using *equivalent* sequences of requests. Two sequences of requests l and l' are equivalent (for type T) if their effects on an object of that type are the same regardless of the state of the object:

$$l \approx_T l' \Leftrightarrow (\forall s,s' \in S_T \; l \in s' -_T s \Leftrightarrow l' \in s' -_T s)$$

The two sequences need not cause the same responses from the object. For the $Stack[X]$ type,

$$\langle push(x),\, pop\rangle \approx \langle\,\rangle$$
$$\langle isEmpty\rangle \approx \langle\,\rangle$$

These equivalences lead to the following stateless compression function c':

$$c'(\langle q_1,\ldots,push(x),pop,\ldots,q_n\rangle) = c'(\langle q_1,\ldots q_n\rangle)$$
$$c'(\langle q_1,\ldots,isEmpty,\ldots,q_n\rangle) = c'(\langle q_1,\ldots q_n\rangle)$$
$$\text{otherwise } c'(\langle q_1,\ldots q_n\rangle) = \langle q_1,\ldots q_n\rangle$$

This stateless compression function is clearly not as effective as the state-sensitive compression function defined above. A problem for future investigation is to identify characteristics of a type that would allow it to have a stateless compression function that is in some sense as good as any state-sensitive compression function.

2.2.7 Distance Functions

In practice, two objects can never be perfectly synchronized at all times, since there is some delay from the time the source object is updated to the time the target object is brought back in synchronization. Applications can often tolerate some synchronization delay.

They may also tolerate some synchronization inaccuracy: the target need not have exactly the same state of the source, just a "close enough" state. The notion of accuracy is common in physical measurements and other numeric values. (How wealthy is Bill Gates? Round to the nearest billion.)

Delay and accuracy are two measures of quality of service for object synchronization, as discussed in Section 2.5. Here, we define a general notion of *distance function* to measure the difference between the states of two objects.

A distance function for type T is a function $d:\{(s,s') \mid s,s' \in S_T \wedge s \rightarrow_T s'\} \rightarrow \mathbf{R}$, where $\mathbf{R}$ is the set of real numbers. It has the following properties for all states s, s', and s'' in S_T such that $s \rightarrow_T s'$ and $s' \rightarrow_T s''$:

$$d(s,s') \geq 0$$
$$d(s,s') = 0 \Leftrightarrow s = s'$$
$$d(s,s'') \leq d(s,s') + d(s',s'') \text{ (the \textit{triangle inequality})}$$

This definition of a distance function d differs from the usual definition in two ways. First, d need not be symmetric, i.e. $d(s,s')$ need not equal $d(s',s)$, because it can be "easier" (in some sense meaningful in the application domain) to change the state of an object of type T from s to s' than the reverse. Second, $d(s,s')$ is not defined for all pairs of states: it need not be defined if it is impossible to change the object's state from s to s', i.e. if $s \not\rightarrow_T s'$.

A distance function is not part of a type definition; there can be more than one distance function defined for the same type.

For a stack, the distance between two states could be defined as the minimum number of *pop* and *push* operations required to change the stack from one state to another. This distance function can be used for any type T:

(4) $d_1(s,s') = \min\{n \mid \langle q_1,\ldots,q_n\rangle \in s' -_T s\}$

It is easy to show that d_1 satisfies the properties of a distance function. In particular, the triangle inequality follows from Equation (3). In the case of the *stack* type, d is symmetric, but this is not true for all types. Equation (5) shows another distance function that could be used with any type T:

(5) $\quad d_2(s,s') = (\mathit{if}\ s = s'\ \mathit{then}\ 0\ \mathit{else}\ 1)$

This distance function is very coarse because it only says whether two states are the same or different.

Both (4) and (5) are examples of a broad class of distance functions of the form:

(6) $\quad d(s, s!!_T\, l) = d'(c(s,l))$

In other words, the distance between two states s and s' is computed by taking *any* sequence of requests l that would change the state of an object of type T, compressing it, and computing the distance from the compressed sequence of requests. The distance functions in Equations (4) and (5) can be expressed as:

$$d_1(s, s!!l) = \mathit{length}(c(s,l))$$
$$d_2(s, s!!l) = (\mathit{if}\ \mathit{length}(c(s,l)) > 0\ \mathit{then}\ 1\ \mathit{else}\ 0)$$

where c is a compression function that produces a minimum-length sequence of requests. A compression function c is said to *support* a distance function d if d can be expressed as in Equation (6).

More refined distance functions can be defined for specific object types; examples are given below.

2.2.8 Examples: Set, Scalar, GeographicLocation

Here are definitions of types $\mathit{Set}[X]$, $\mathit{Scalar}[X]$, and $\mathit{GeographicLocation}$.

Type $\mathit{Set}[X]$ defines an object whose state is a set of values drawn from some domain X. The state domain of $\mathit{Set}[X]$ is $\mathbf{P}(X)$, the power set (set of all subsets) of X. The definition is trivial because it relies on commonly-understood mathematical notation.

$$S = \mathbf{P}(X)$$
$$Q = \{\mathit{insert}(x) \mid x \in X\} \cup \{\mathit{delete}(x) \mid x \in X\} \cup \{\mathit{member}(x) \mid x \in X\}$$
$$R = \{\mathit{ok}, \mathit{missingElement}, \mathit{true}, \mathit{false}\}$$
$$SQ = S \times Q$$
$$\varepsilon = \varnothing$$

$$s!\mathit{insert}(x) = s \cup \{x\}$$
$$s?\mathit{insert}(x) = \mathit{ok}$$

$s!delete(x) = s - \{x\}$
$s?delete(x) = (if\ x \in s\ then\ ok\ else\ missingElement)$

$s!member(x) = s$
$s?member(x) = (x \in s)$

The $Set[T]$ type has a number of equivalent request sequences:

$\langle insert(x), delete(x) \rangle \approx \langle\ \rangle$
$\langle delete(x), insert(x) \rangle \approx \langle insert(x) \rangle$
$\langle insert(x), insert(y) \rangle \approx \langle insert(y), insert(x) \rangle$
$\langle insert(x), delete(y) \rangle \approx \langle delete(y), insert(x) \rangle$ for $x \neq y$
$\langle delete(x), delete(y) \rangle \approx \langle delete(y), delete(x) \rangle$
$\langle member(x) \rangle \approx \langle\ \rangle$

A reasonable distance function is the minimum number of *insert* and *delete* requests needed to change a $Set[X]$ object from state s to state s' (Equation (4)).

The type $Scalar[X]$ provides *set* and *get* operations for scalar values of any domain X. X could be the set of integers $\mathbf{Z}$, the set of reals $\mathbf{R}$, the set of character strings, etc. It could also be a set of "larger" values such as images, for which *set* and *get* operations are sufficient. Assume that for any domain X there is a distinguished value $x_0 \in X$ that is the initial state for objects of this type. The type is defined as follows:

$S = X$
$Q = \{set(x) \mid x \in X\} \cup \{get\}$
$R = \{ok\} \cup X$
$SQ = S \times Q$
$\varepsilon = x_0$

$s!set(x) = x$
$s?set(x) = ok$

$s!get = s$
$s?get = s$

The $Scalar[X]$ type has two equivalent request sequences:

$\langle set(x), set(y) \rangle \approx \langle set(y) \rangle$
$\langle get \rangle \approx \langle\ \rangle$

As a result of these equivalences, the distance functions defined by Equations (4) and (5) are equivalent, and not very informative. More informative distance functions can be defined for specific domains. For instance, if D is $\mathbf{Z}$ or $\mathbf{R}$, the absolute value $d(s,s') = | s' - s |$ is more useful. (Here, $s' - s$ denotes subtraction, not state difference as defined in Equation (2)).

The *GeographicLocation* type models points on the Earth identified by latitude and longitude. The GPS sensor in Figure 6 is an example of an object of this type.

$$
\begin{aligned}
&S = \{(lat,long) \mid lat,long \in \mathbf{R} \wedge -90 \le lat \le 90 \wedge -180 \le long \le 180\} \\
&Q = \{setPos(lat',long') \mid lat',long' \in \mathbf{R} \wedge -90 \le lat' \le 90 \wedge -180 \le long' \le 180\} \\
&\quad \cup \{move(lat',long') \mid lat',long' \in \mathbf{R} \wedge -90 \le lat' \le 90 \wedge -180 \le long' \le 180\} \\
&\quad \cup \{getPos\} \\
&R = \{ok\} \cup \{(lat',long') \mid lat',long' \in \mathbf{R} \wedge -90 \le lat' \le 90 \wedge -180 \le long' \le 180\} \\
&SQ = S \times Q - \{((lat,long), move(lat',long')) \mid lat + lat' \le -90 \vee lat + lat' \ge 90\}
\end{aligned}
$$

$$
\begin{aligned}
&(lat,long)!setPos(lat',long') = (lat',long') \\
&(lat,long)?setPos(lat',long') = ok
\end{aligned}
$$

$$
\begin{aligned}
&(lat,long)!move(lat',long') = (lat + lat', (long + long') \bmod 360 - 180) \\
&(lat,long)?move(lat',long') = ok
\end{aligned}
$$

$$
\begin{aligned}
&(lat,long)!getPos = (lat,long) \\
&(lat,long)?getPos = (lat,long)
\end{aligned}
$$

A reasonable distance function for this type is the great circle distance.

2.2.9 Object Containment, Identifiers, and Creation

Real-world object-oriented applications make heavy use of object containment, identifiers, and creation. The Active Views mechanisms—view, history, and synchronization—must work in these settings. In particular, it must be possible to define a *view* of a collection of related objects, create a *history* of a collection of related objects, and *synchronize* the states of two such collections, where client programs may add, remove, and operate on objects in the collection.

Here are the principles of our model of object containment, identifiers, and creation:

1. An object O (the *container*) can contain another object O' (the *contained* object), meaning that the state of O' is part of the state of O, and any change to the state of O' is also a change to the state of O.
2. A container can have multiple contained objects. The number of contained objects may be fixed, determined by the container's type definition. Alternatively, the container's type definition can include one or more requests for creating and destroying contained objects. All objects are created by sending a request to some container. (The first object, the container for all other objects in a system, must be created in some other manner.)
3. A client application sends a request to a contained object indirectly via its container. The container's type may provide different forms of request for accessing different contained objects. Alternatively, the container's type may provide a request of the form $invoke(objectIdentifier, request)$, where $objectIdentifier$ is a value, meaningful to the container, that distinguishes among the contained objects and $request$ is a request defined by the contained object's type.

4. An object O'' can have, as part of its state, an object identifier for an object O' that is contained by O. This is a *reference* from O'' to O'. O'' does not contain O'.
5. Containment is a *behavior,* not a structural relationship among objects. In other words, a container, by its behavior in response to requests, provides the *appearance* that it contains objects. Views can be constructed that provide *different containment relationships* among a set of objects that reference each other.

Note that an object identifier is meaningful to a particular container. The object-oriented literature abounds with assertions that object identifiers must be *globally* unique. True global uniqueness cannot be ensured; there is always some context (perhaps implicit) within which the uniqueness is ensured, such as an address space, an OODBMS, or a networking environment. This context acts as the containing object.

There are various kinds of containers—sets, arrays, records, etc.—so there is no single *container* object type. However, the following examples illustrate our model.

2.2.10 Example: Record

A record has a fixed number of contained objects that can be of different types. The contained objects are distinguished by field names rather than object identifiers. Here, we define a type $Record[f_1, T_1, \ldots, f_n, T_n]$ that has fields $f_1, \ldots f_n$ of types $T_1, \ldots, T_n$ respectively.

$$S = S_{T_1} \times \ldots \times S_{T_n}$$
$$Q = \bigcup_{1 \le i \le n} \{invoke\_f_i(q') \mid q' \in Q_{T_i}\}$$
$$R = \bigcup_{1 \le i \le n} R_{T_i}$$
$$SQ = \bigcup_{1 \le i \le n} \{((s_1, \ldots, s_n), invoke\_f_i(q')) \mid (s_i, q') \in SQ_{T_i}\}$$
$$\varepsilon = (\varepsilon_{T_1}, \ldots, \varepsilon_{T_n})$$
$$(s_1, \ldots, s_i, \ldots s_n)!invoke\_f_i(q') = (s_1, \ldots, s_i !_{T_i} q', \ldots s_n)$$
$$(s_1, \ldots, s_i, \ldots s_n)?invoke\_f_i(q') = s_i ?_{T_i} q'$$

The type has a set of equivalent request sequences that reflect the independence of the record's fields, plus equivalent request sequences derived from the underlying types:

$$\langle invoke\_f_i(q'), invoke\_f_j(q'')\rangle \approx \langle invoke\_f_j(q''), invoke\_f_i(q')\rangle \text{ for } i \ne j$$
$$\langle q', q''\rangle \approx_{T_i} \langle q'', q'\rangle \Rightarrow \langle invoke\_f_i(q'), invoke\_f_i(q'')\rangle \approx \langle invoke\_f_i(q''), invoke\_f_i(q')\rangle$$

If $d_1, \ldots, d_n$ are distance functions for the underlying field types, then a reasonable distance function for the $Record[f_1, T_1, \ldots f_n, T_n]$ type is:

$$d((s_1, \ldots, s_n), (s'_1, \ldots, s'_n)) = \sum_{1 \le i \le n} d_i(s_i, s'_i)$$

Variations on this distance function (root mean square, maximum, weighted sum, etc.) are also possible.

2.2.11 Example: SetContainer

An object of type $SetContainer[T,I]$ contains other objects of type T. Identifiers for the contained objects are drawn from some domain I. The state domain S for type $SetContainer[T,I]$ is a partial function from I to the state space S_T of T:

$$S = I \nrightarrow S_T$$

Initially, the function is completely undefined, meaning that the container is empty:

$$\varepsilon = \varnothing$$

There are requests for inserting (creating) a new contained object, deleting of an existing one, testing for membership, and invoking an operation on an existing contained object.

$$\begin{aligned}
Q &= \{insert(i) \mid i \in I\} \\
&\quad \cup \{delete(i) \mid i \in I\} \\
&\quad \cup \{member(i) \mid i \in I\} \\
&\quad \cup \{invoke(i,q) \mid i \in I \wedge q \in Q_T\} \\
R &= \{ok, missingElement, true, false\} \\
&\quad \cup \{response(r) \mid r \in R_T\} \\
SQ &= S \times Q
\end{aligned}$$

The $insert(i)$ request adds an object with initial state ε_T to the container and assigns it object identifier i. If an object with that identifier is already in the container, it is replaced by an object with this initial state. The *overriding union* operator $\uplus$ accomplishes this replacement.

$$\begin{aligned}
s!insert(i) &= s \uplus \{(i, \varepsilon_T)\} \\
s?insert(i) &= ok
\end{aligned}$$

The $delete(i)$ request removes the object with identifier i from the container, if it exists. Here, $\mathrm{dom}(s)$ is the domain of the partial function, which is the set of identifiers of the objects in the container.

$$\begin{aligned}
s!delete(i) &= (\mathit{if}\ i \in \mathrm{dom}(s)\ \mathit{then}\ s - \{(i, s(i))\}\ \mathit{else}\ s) \\
s?delete(i) &= (\mathit{if}\ i \in \mathrm{dom}(s)\ \mathit{then}\ ok\ \mathit{else}\ missingElement)
\end{aligned}$$

The $member(i)$ request tests whether the container has an object with identifier i.

$$\begin{aligned}
s!member(i) &= s \\
s?member(i) &= (i \in \mathrm{dom}(s))
\end{aligned}$$

Finally, the $invoke(i,q)$ request sends request q to the contained object with identifier i, assuming it is in the container. The state of the contained object, which is part of the state of the container object, changes as specified by the definition of type T. The response is also as specified by type T, except the response value $s(i)?_T\, q$ from the contained object is wrapped in the form $response(s(i)?_T\, q)$. This guarantees that the response from the contained object—even if it is *missingElement*—cannot be confused with a response *missingElement* from the container. The contained object could return the response *missingElement* if, for instance, $T = Set[X]$.

$$s!invoke(i,q) = (\mathit{if}\ i \in \mathrm{dom}(s)\ \mathit{then}\ s \mathbin{\psi} \{(i, s(i)!_T\, q)\}\ \mathit{else}\ s)$$
$$s!invoke(i,q) = (\mathit{if}\ i \in \mathrm{dom}(s)\ \mathit{then}\ response(s(i)?_T\, q)\ \mathit{else}\ missingElement)$$

This type has a number of request equivalences. The first several correspond to the equivalences for type $Set[X]$. As usual, $\approx$ is shorthand for $\approx_{SetContainer[T,I]}$.

$$\langle insert(i), invoke(i,q_1),\ldots,invoke(i,q_n), delete(i)\rangle \approx \langle\ \rangle$$
$$\langle delete(i), insert(i)\rangle \approx \langle insert(i)\rangle$$
$$\langle insert(i), insert(j)\rangle \approx \langle insert(j), insert(i)\rangle$$
$$\langle insert(i), delete(j)\rangle \approx \langle delete(j), insert(i)\rangle \text{ for } i \neq j$$
$$\langle delete(i), delete(j)\rangle \approx \langle delete(j), delete(i)\rangle$$
$$\langle member(i)\rangle \approx \langle\ \rangle$$

The following three equivalences show that invoking an operation on one contained object doesn't interfere with inserting, deleting, or operating on another contained object:

$$\langle insert(i), invoke(j,q')\rangle \approx \langle invoke(j,q'), insert(i)\rangle \text{ for } i \neq j$$
$$\langle invoke(i,q), invoke(j,q')\rangle \approx \langle invoke(j,q'), invoke(i,q)\rangle \text{ for } i \neq j$$
$$\langle invoke(i,q), delete(j)\rangle \approx \langle delete(j), invoke(i,q)\rangle \text{ for } i \neq j$$

Finally, for every equivalence $\langle q_1,\ldots,q_n\rangle \approx_T \langle q_1',\ldots,q_m'\rangle$ for type T, there is a corresponding equivalence for type $SetContainer[T,I]$:

$$\langle invoke(i,q_1),\ldots,invoke(i,q_n)\rangle \approx_{SetContainer(T,I)} \langle invoke(i,q_1'),\ldots,invoke(i,q_m')\rangle$$

Any number of distance functions can be defined for type $SetContainer[T,I]$. A plausible distance function d can be constructed using any distance function d' defined for type T. For any two states of the container, this distance function sums the number of object insertions required, the number of object deletions required, and the distances (using d') between the original and final states of each object that is in the container in both states:

$$\begin{aligned} d(s,s') = {} & |\mathrm{dom}(s') - \mathrm{dom}(s)| + |\mathrm{dom}(s) - \mathrm{dom}(s')| \\ & + \sum_{i \in \mathrm{dom}(s) \cap \mathrm{dom}(s')} \min(d'(s(i), s'(i)), 1 + d'(\varepsilon_T, s'(i))) \end{aligned}$$

(The added complexity in the third term is to account for the possibility it is cheaper to overwrite an existing object with a new object having the same identifier i, and to bring that new object to state $s'(i)$, than it is to bring the existing object from state $s(i)$ to state $s'(i)$).

2.2.12 *Objects, Object Identifiers and References, Object Containment, and Values*

It is worth drawing clear distinctions between an object and a value, between an object and an object identifier or reference, and between an object reference and object containment. These distinctions are often muddied in the literature but are vital to precise definitions of Active Views mechanisms.

Consider an object O_1 of type $SetContainer[T,I]$, an object O_2 of type $Set[I]$, and an object O_3 of type $Set[S_T]$. O_1 *contains* a set of *objects*, each of whose states is a *value* that is member of S_T. O_2 is a set of *identifiers*. These identifiers are *references* to objects contained in O_1. O_2 references these objects, but does not contain them. O_3 is a set of *values*, each of which is a member of S_T. It is possible to send a request to a member of O_1, but not to a value in O_3. More generally,

- A *value* is an element of some mathematical set or domain.
- An *object* is a software construct that has a state. It receives requests, modifies its state, and returns responses as defined by its type. The request, response and the state of an object are all values.
- An *object identifier* is a value, meaningful to a *container object*, that distinguishes among contained objects.
- An *object reference* is a value (specifically, an object identifier for a *referenced* object) that is part of the state of a *referring* object.

The $SetContainer[T,I]$ example illustrates the first four principles of our model of object containment, identifiers, and creation listed above. The fifth principle, that containment is behavior, not structure, and that different containment relationships can be constructed using views, may be surprising. Figure 7 shows how this can be done. Let O be an object of type $SetContainer[T,I]$ that is a view $O = SubsetContainer(O_1, O_2)$, where objects O_1 and O_2 are as defined above. A precise specification of the $SubsetContainer$ view will be given in Section 2.3.2. Basically, the view function $SubsetContainer$ causes O to contain the objects that are contained by O_1 *and* referenced by O_2. In other words, if object O' is contained in O_1 and is referenced by O_2, it is also contained in O. If the state of O' changes because an *invoke* request is sent to O_1, the state change is apparent in O also, and vice versa. As object identifiers are inserted into O_2 or deleted from it, the set of objects contained by O changes correspondingly. The net effect is that O contains the objects that O_2 merely references.

The ability to construct different containment relationships provides significant flexibility for the Active Views mechanisms, which are based on object states. For instance, a synchronized copy of O_2 would track the insertions and deletions of object references in O_2, while a synchronized copy of O would track insertions, deletions, *and* updates to the states of the referenced objects. Either outcome might be desirable based on application requirements.

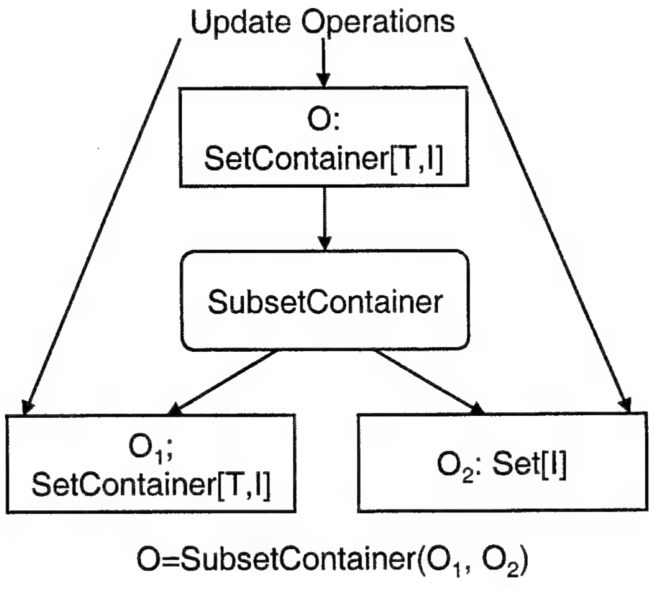


Figure 7. Constructing a Containment Relationship Using a View

2.2.13 Generating Object Identifiers

There are two general approaches to generating object identifiers: the client provides the identifier as part of the request that it sends to the container to create the object, or the container generates the identifier as part of the creation process and returns it to the client in the response. *SetContainer* uses the first approach; the latter approach is more typical in object-oriented systems, which often provide one or more *constructors* for each object type.

As with any object type, a container type must completely specify the container object's behavior. This includes the assignment of object identifiers as new contained objects are created. If the client provides the identifier in the request, this requirement is automatically fulfilled. If the container generates the identifier, care must be taken to ensure that the identifier generated is completely predicable from the container's type and the sequence of requests sent to the container so far. Otherwise, the Active Views mechanisms—in particular object synchronization—will not function properly: a request *invoke(objectIdentifier, request)*, when sent to both the source and target container objects, may not have the same effect because *objectIdentifier* doesn't reference the same contained object in both cases. Most object-oriented environments—language run-time environments, ORBs, and OODBMSs—don't have predictable object identifier generation, so these "containers" are not suitable source and target objects for object synchronization.

It is possible to use some scheme to track corresponding objects in the source and target containers. During synchronization, the scheme would have to intercept and translate all object references from the set used by the source container to the set used by the target container. The scheme would have to be tailored to the specific container object type, and would make implementation of a type-independent synchronization scheme difficult.

2.3 Views

The view concept originated in database management systems and has been adopted in other contexts, such as computer-aided design tools, where multiple perspectives of a body of information are needed. Views have the following uses:

- *Meeting clients' information needs*—Providing client applications and users with the information they need, but no more, from the mass of accessible data.
- *Management of system evolution*—Systems, and the objects that implement them, change over time. A client application or user can create a view of these objects that meets its information needs. As the system evolves over time, the view insulates the client from irrelevant changes to the underlying objects, thus simplifying system evolution. This approach to managing system evolution was recognized long ago in the ANSI/SPARC three-schema architecture [TK77], but seems to get little attention today.
- *Integration of legacy applications and databases*—Legacy applications and databases typically require and/or provide information in a different format than the one used by a new system into which they are being integrated. Views can be used to accomplish the desired transformations.
- *Information security*—Views have long been used in relational databases to define the restricted subset of a database that a client may access.

In relational databases, a view is a table whose state is defined using a fixed set of relational operations over tables and scalar operators over column values. In object oriented database management systems, as represented by the ODMG standard [CATT97] or experimental systems, e.g. MultiView [KR98], a view can be defined using a fixed set of operators over collections (sets, sequences, bags, arrays, etc) and arbitrary computed functions over scalar values.

Our goal here is to have a very general definition of views for object-oriented systems. This generalization of the view concept allows the benefits of views, listed above, to be realized in a much broader range of applications. A *view* is an object O whose state is a function of the states of *source objects* $O_1,\ldots,O_n$. Informally, the functional relationship is denoted $O = F(O_1,\ldots,O_n)$, where F is a *view function*. More precisely, the functional relationship is between the *states* of these objects. The source and view objects can be of arbitrary type. The view function is also arbitrary: it can perform any kind of information selection, abstraction, or transformation.

When a client application sends a request to a source object that causes a state change, the view object's state changes to maintain the functional relationship $O = F(O_1,\ldots,O_n)$. Similarly, when a client application sends a request to the view object that causes a state change, one or more source objects' states change to maintain the functional relationship. Clients cannot perceive these state changes directly, but only by the responses that the source and view objects return in response to requests.

A *view specification* is a definition of a functional relationship between a view object and a set of source objects, and a definition of how that relationship must be maintained when requests are sent to these objects. A *view implementation* is a procedure that realizes a view specification. Figure 8 shows a graphical notation for a view specification V and a view implementation P. The arrows in the view specification indicate the direction of the functional relationship, and the dotted lines indicate that this is a specification of behavior, not an implementation. The arrows in the view implementation indicate the direction in which requests are sent: a request sent to view object O is sent to procedure P, which sends one or more requests to each of the source objects. O is a virtual view: its state is maintained in its source objects. Materialized views [GM95], in which a view's state is maintained in an object separate from the source object, are also possible. In general, a view specification can be implemented in more than one way.

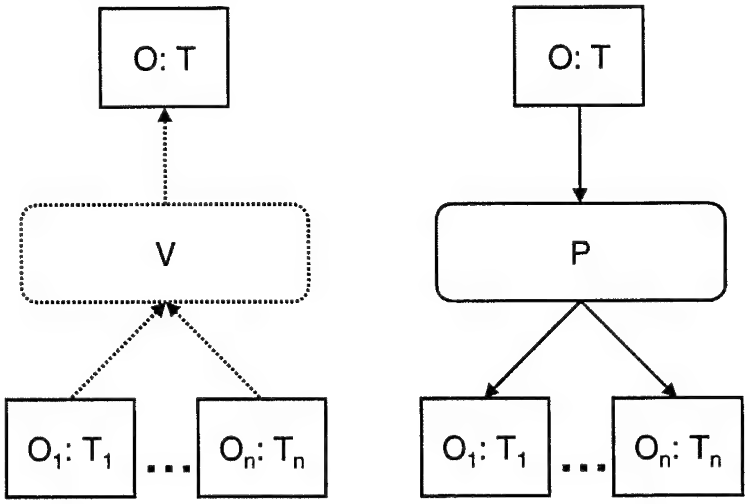


Figure 8. Graphical Representation of a View Specification and a View Implementation

We retain a key characteristic of views: *they can be composed.* The ability to compose views means that a user can define complex views out of simpler ones drawn from a view library, as in Figure 6.

The ability to compose views also means that algebraic properties of the view functions can be exploited to improve system performance. This has been done extensively in database query optimization, and is an area for future investigation in this more general context.

Views are essential to our approach for active information systems. In our approach, an event is a change in state of some object. *A user or application defines events of interest by creating a view whose state changes are those events.* Here, the ability to compose views simplifies the task of defining the events of interest.

We begin with a definition of a view specification and an example. We then discuss various ways of implementing a view specification in software. The remainder of the section covers further topics about views, including how views relate to the common concepts of object implementation and type inheritance.

2.3.1 View Specifications

A view specification V is a 5-tuple $((T_1^V,\ldots,T_n^V),T^V,F^V,I^V,U^V)$. The superscripts are used to distinguish elements of view specification V from those of different view specifications, and will be dropped where there is no ambiguity.

- $(T_1,\ldots,T_n)$ are the source object types.
- T is the view object type.
- F is the view function. It has signature $S_{T_1}\times\ldots\times S_{T_n}\rightarrow S_T$. The view function maps the state domains of the source objects $O_1,\ldots,O_n$ to the state domain of the view object O. The relationship $s=F(s_1,\ldots,s_n)$ is *invariant*, guaranteed to hold at all times except when a source object or the view object has received a request and has not yet replied.
- SI is the *source invariant*. It is a predicate over $S_{T_1}\times\ldots\times S_{T_n}$ that constrains the combined states of the source objects. It must be true at all times except during request processing. As a degenerate case, I may be identically true.
- U is the *update constraint*. It is a predicate over $S_{T_1}\times\ldots\times S_{T_n}\times S_{T_1}\times\ldots\times S_{T_n}\times Q_T$ that constrains how the source objects are updated when a client sends a request to the view object. The update constraint is needed because F by itself cannot determine the new states of the source objects unless it is one-one. However, this constraint need not *completely* specify how the source objects are updated. An example of a partial specification is given below.

In some applications, it isn't meaningful or desirable for clients to update the state of the view object directly, but only indirectly by sending requests to the source objects. The opposite situation also occurs, where it isn't meaningful or desirable for clients to update the states of the source objects directly, but only indirectly by sending requests to the view object. A view specification is called *forward-updatable* if clients may update the source objects directly, and *reverse-updatable* if clients may update the view object directly. A view specification can be forward updatable, reverse updatable, or both.

If the view specification is forward-updatable, each client that updates a source object must ensure that the source invariant is restored. If the view is reverse-updatable, the view implementation must ensure that the source invariant is restored following an update to the view object, and that the update constraint is observed.

Like a type definition, a view specification can be parameterized. This often occurs because the source and view object types are themselves parameterized, but can also be done to parameterize the functional relationship between the source and view objects. A parameterized view specification is denoted $V[p_1,\ldots,p_n]$.

A view specification must meet two conditions to ensure it is consistent with the definitions of the source and object types. The first condition applies to forward updating: any legal update to a source object must cause a legal state transition in the view object:

(7) $$\begin{array}{l} \forall s_1 \in S_{T_1}, \ldots, s_n \in S_{T_n}, q \in Q_{T_i} \\ \quad SI(s_1, \ldots, s_n) \wedge (s_i, q) \in SQ_{T_i} \wedge SI(s_1, \ldots, s_{i-1}, s_i !_{T_i} q, s_{i+1}, \ldots, s_n) \\ \quad \Rightarrow F(s_1, \ldots, s_n) \rightarrow_T F(s_1, \ldots, s_{i-1}, s_i !_{T_i} q, s_{i+1}, \ldots, s_n) \end{array}$$

As stated earlier, a client application that updates a source object is responsible for maintaining the source invariant. Therefore we need only consider those source updates for which the source invariant is true both before and after the update occurs.

The second condition applies to reverse updating: any legal update to the view object must be explainable in terms of legal updates to the source object:

(8) $$\begin{array}{l} \forall s_1 \in S_{T_1}, \ldots, s_n \in S_{T_n}, q \in Q_T \; SI(s_1, \ldots, s_n) \wedge (F(s_1, \ldots, s_n), q) \in SQ_T \\ \quad \Rightarrow \exists s_1' \in S_{T_1}, \ldots, s_n' \in S_{T_n} \\ \quad\quad \forall i \; s_i \rightarrow_{T_i} s_i' \wedge F(s_1, \ldots, s_n) !_T q = F(s_1', \ldots, s_n') \wedge SI(s_1', \ldots, s_n') \wedge U(s_1, \ldots, s_n, s_1', \ldots, s_n', q) \end{array}$$

As stated earlier, we must ensure that these updates restore the source invariant and observe the update constraint. There may be more than one set of updates to the source objects that meet this requirement; a view implementation is free to choose any of them.

This definition of view updates allows the source objects to change state even if the request sent to the view object does not change the view's state. The example below illustrates this.

2.3.2 Example: SubsetContainer

Here is a formal specification of the $SubsetContainer[T, I]$ view introduced in Section 2.2.12 to illustrate construction of a container from a set of object references. This view defines a relationship $O = SubsetContainer(O_1, O_2)$ between view object O and source objects O_1 and O_2 having the following types:

$$\begin{array}{l} T_1^{SubsetContainer[T,I]} = SetContainer[T, I] \\ T_2^{SubsetContainer[T,I]} = Set[I] \\ T^{SubsetContainer[T,I]} = SetContainer[T, I] \end{array}$$

(For brevity, we will omit the view parameters when they are clear.)

The view function $F^{SubsetContainer[T,I]}$ defines the state of the view object O in terms of the state of the source objects. Based on the types of the source and view objects, the signature of $F^{SubsetContainer[T,I]}$ is:

$$F^{SubsetContainer[T,I]} : (I \nrightarrow S_T) \times \mathbf{P}(I) \rightarrow (I \nrightarrow S_T)$$

An object is in container O if and only if it is in container O_1 *and* a reference to it is in set O_2. The state of each object in container O is the same as the state of the corresponding object in container O_1. In mathematical terms, if s_1 and s_2 are the states of O_1 and O_2 respectively, then the state of O is the partial function derived from s_1 by eliminating elements of the domain that aren't in s_2:

$$F^{SubsetContainer[T,I]}(s_1,s_2) = \{(i,x) \mid (i,x) \in s_1 \wedge i \in s_2\}$$

For this view, we choose not to impose any further constraints on the source objects' states via a source invariant $SI^{SubsetContainer[T,I]}$. In other words, $SI^{SubsetContainer[T,I]}$ is identically true. So, for instance, there can be object references in O_2 for which there are no corresponding objects in container O_1.

The update constraint $U^{SubsetContainer[T,I]}(s_1,s_2,s_1',s_2',q)$ is stated as a set of four rules, one for each form of request q for type $SetContainer[T,I]$. According to the definition of type $SetContainer[T,I]$, an *insert(i)* request causes any existing object with identifier i to be overwritten by a new object having the same identifier:

$$\begin{aligned} q = insert(i) \Rightarrow\ & s_1' = s_1 \uplus \{(i,\varepsilon_T)\} \\ & \wedge s_2' = s_2 \cup \{i\} \end{aligned}$$

For a *delete(i)* request, we can delete the object from O_1, delete the reference from O_2, or delete both. We choose only to delete the reference from O_2; perhaps other applications still have use for object i in container O_1:

$$\begin{aligned} q = delete(i) \Rightarrow\ & s_1' = s_1 \\ & \wedge s_2' = s_2 - \{i\} \end{aligned}$$

A *member(i)* request doesn't affect the state of the view object O, so neither source object must change state. However, if i is a dangling reference in O_2 to an object that doesn't exist in O_1, the view implementation *may* remove it. This is an example of a partial specification of a view's update behavior.

$$\begin{aligned} q = member(i) \Rightarrow\ & s_1' = s_1 \\ & \wedge s_2 - (\{i\} - \mathrm{dom}(s_1)) \subseteq s_2' \subseteq s_2 \end{aligned}$$

Finally, an *invoke(i,q)* request to the view object O is passed to the source object O_1 unless there is no such object in the view. Again, deletion of a dangling reference is permitted but not required:

$$\begin{aligned} q = invoke(i,q') \Rightarrow\ & s_1' = (if\ i \in \mathrm{dom}(s_1) \cap s_2\ then\ s_1 \uplus \{(i, s_1(i)!_T\, q')\}\ else\ s_1) \\ & \wedge s_2 - (\{i\} - \mathrm{dom}(s_1)) \subseteq s_2' \subseteq s_2 \end{aligned} \tag{9}$$

Now let's see show that Equation (7), which deals with forward updates to the view, is satisfied. This equation requires that any legal update to either source object must cause a legal state transition in the view object. Let s_1, s_2, and s be the states of the source and view objects before the request is sent, and s_1', s_2', and s' be the states after the request. The proof strategy is to show that $s' = s!!_{SetContainer}\langle q_1, \ldots q_n \rangle$, for some sequence of requests $\langle q_1, \ldots q_n \rangle$ applied to the view object, by (1) expressing s' in terms of s_1' and s_2' using the view function definition, (2) expressing s_1' and s_2' in terms of s_1 and s_2 using the type definitions for O_1 and O_2, and (3) expressing $s!!_{SetContainer}\langle q_1,\ldots q_n\rangle$ in terms of s_1 and s_2 using the view function definition and the type definition of O.

The most complex case is an $invoke(i,q')$ request sent to O_1. There are three subcases.

Subcase 1: $i \in dom(s_1) \wedge i \in s_2$. In this case, O changes state as if an $invoke(i,q)$ request were sent to it:

$$\begin{aligned}
s' &= \{(j,x) \mid (j,x) \in s_1' \wedge j \in s_2'\} \\
&= \{(j,x) \mid (j,x) \in s_1 \uplus \{(i, s_1(i)!_T\, q')\} \wedge j \in s_2\} \\
&= \{(j,x) \mid (((j,x) \in s_1 \wedge j \neq i) \vee (j,x) = (i, s_1(i)!_T\, q')) \wedge j \in s_2\} \\
&= \{(j,x) \mid ((j,x) \in s_1 \wedge j \in s_2 \wedge j \neq i) \vee ((j,x) = (i, s_1(i)_T\, q') \wedge j \in s_2)\} \\
&= \{(j,x) \mid ((j,x) \in s_1 \wedge j \in s_2 \wedge j \neq i) \vee (j,x) = (i, s_1(i)_T\, q')\} \\
&= \{(j,x) \mid ((j,x) \in s \wedge j \neq i) \vee (j,x) = (i, s(i)_T\, q')\} \\
&= s \uplus \{(i, s(i)_T\, q')\} \\
&= s!!_{SetContainer}\langle invoke(i,q')\rangle
\end{aligned}$$

Subcase 2: $i \in dom(s_1) \wedge i \notin s_2$. In this case, O_1 changes state but O does not:

$$\begin{aligned}
s' &= \{(j,x) \mid (j,x) \in s_1' \wedge j \in s_2'\} \\
&= \{(j,x) \mid (j,x) \in s_1 \uplus \{(i, s_1(i)!_T\, q')\} \wedge j \in s_2\} \\
&= \{(j,x) \mid (((j,x) \in s_1 \wedge j \neq i) \vee (j,x) = (i, s_1(i)!_T\, q')) \wedge j \in s_2\} \\
&= \{(j,x) \mid ((j,x) \in s_1 \wedge j \in s_2 \wedge j \neq i) \vee ((j,x) = (i, s_1(i)_T\, q') \wedge j \in s_2)\} \\
&= \{(j,x) \mid (j,x) \in s_1 \wedge j \in s_2\} \\
&= s!!_{SetContainer}\langle\,\rangle
\end{aligned}$$

Subcase 3: $i \notin dom(s_1)$. In this case, neither O_1 nor O changes state. The proof is trivial.

Proofs for other requests to O_1 and O_2 are similar.

Finally, let's show that Equation (8), which deals with reverse updating, is satisfied. This equation requires that any legal update to the view object must be explainable in terms of legal updates to the source objects that preserve the source invariant and observe the update constraint. The proof strategy is to (1) consider different forms of request to the view object one at a time, (2) for a particular request form, identify a sequence of corresponding requests to each of the source objects, (3) show that these requests maintain the source invariant and observe the view update invariant, and (4) verify that the new source object states are consistent with the new view object state as defined by the view object's state transition function.

As an example, consider the request $invoke(i,q')$ sent to O. Again, there are subcases depending on the value of i. The most important subcase is $i \in \mathrm{dom}(s_1) \wedge i \in s_2$. In this case, the sequences of requests for O_1 and O_2 are $\langle invoke(i,q')\rangle$ and $\langle\,\rangle$ respectively, so that:

$$s_1' = s_1 !!_{SetContainer(T)} \langle invoke(i,q') \rangle = s_1 \Psi \{(i, s_1(i)_T q')\}$$
$$s_2' = s_2 !!_{Set(I)} \langle\, \rangle = s_2$$

The source invariant $SI^{SubsetContainer[T,I]}$ is maintained, since it is identically true. The update constraint (9) is also clearly satisfied. The proofs are similar for the remaining subcases for request $invoke(i,q')$, and for the other request forms that can be sent to view object O.

2.3.3 View Implementations and Object Implementations

As stated earlier, a view implementation is a software realization of a view specification. It is a procedure that receives requests from clients and generates responses according to the view object's type. Figure 9 shows how a view implementation operates. Between receiving the request and sending the response, the view implementation can send requests to the source objects $O_1,\ldots,O_n$ and use the source objects' responses to compute its response to the client. It can use temporary storage while computing its response, but cannot maintain state between requests except in its source objects.

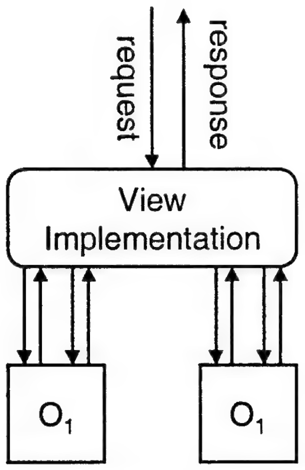


Figure 9. View Implementation Operation

The view implementation is constrained to interact with the source objects using only the requests defined by the source objects' types. It cannot access the source objects' states directly. It is further constrained to obey the view specification's update constraint.

Here is a sketch of an implementation of the *SubsetContainer*$[T,I]$ view specification.

- In response to an $insert(i)$ request, the implementation sends an $insert(i)$ request to both source objects as required by the update constraint, and returns the response ok.
- In response to a $delete(i)$ request, it sends $member(i)$ requests to both source objects to determine whether $i \in \text{dom}(s_1) \cap s_2$. If not, its response is $missingElement$ as required by the view object's type; otherwise the response is ok. In either case, it sends a $delete(i)$ request to O_2 as required by the update constraint.

- In response to a *member*(i) request, it determines whether $i \in \mathrm{dom}(s_1) \cap s_2$ as above and returns the appropriate response. If i is a dangling reference in O_2, it sends O_2 a *delete*(i) request as allowed by the update constraint.
- In response to an *invoke*(i,q) request, it again determines whether $i \in \mathrm{dom}(s_1) \cap s_2$, and if so sends an *invoke*(i,q) request to O_1 and returns O_1's response to the client. Otherwise, it sends a *missingElement* response to the client.

A view implementation looks suspiciously like the conventional notion of object class or object implementation. This is true: *an object implementation is a form of view implementation*, where the view object type is the object type being implemented, the view implementation is the class code, and the source objects hold the implementation state. In this context, the view function serves as the abstraction function and the source invariant serves as the implementation invariant [MEYE97]. The view specification is normally not forward updatable, since direct updates to the implementation state would violate encapsulation.

2.3.4 *Indirect View Implementations and Inheritance*

The implementation of the *SubsetContainer*$[T,I]$ view specification sketched above is a *direct* implementation: it is a single procedure that realizes the required behavior of the view object by interactions with the source objects.

Sometimes a view specification has no direct implementation, due to the constraints that the source objects' types impose on access to their states. Consider the forward-updatable view specification $N = \mathit{SizeOf}(S)$, where view object N has type *Scalar*[N] (a non-negative integer object) and source object S has type *Set*$[X]$ for some domain X. When a client sends a *get* request to N, the response is the size of S. How can this view specification be implemented? The *Set*$[X]$ type provides no means by which a client could determine the elements of the set, except by exhaustively sending S a *member*(x) request for each element of the domain X and count the number of *true* responses. If X is infinite, this is impossible; for man common finite domains (integers, floats, etc.) it is unacceptably inefficient.

In such cases, it is legitimate to revise the type specifications of the source objects to provide better access to the source objects' states. Type specifications should be refined where appropriate to meet legitimate application needs. Another approach is to use an *indirect* view implementation, as shown in Figure 10. In this approach, we assume that the set S is implemented as a linked list. This implementation has two implementation objects: *ListHead*, which holds a reference to the first element of the list, and *ListElements*, which is a container of list elements. The *SetAsList* procedure implements the set abstraction in the usual way. The integer object N is a view of the same two implementation objects, using a different procedure *ListTraverse* to count the number of elements in the list.

As another example, relational DBMSs use indirect view implementations to enforce the functional relationship between a view (derived) table and the base tables from which it is defined. For instance, view V might be defined as the join of two base tables, $V = T_1 \bowtie T_2$. In fact, T_1 and T_2 are direct views of the underlying physical storage structures that implement them, and the DBMS computes V from these structures as well.

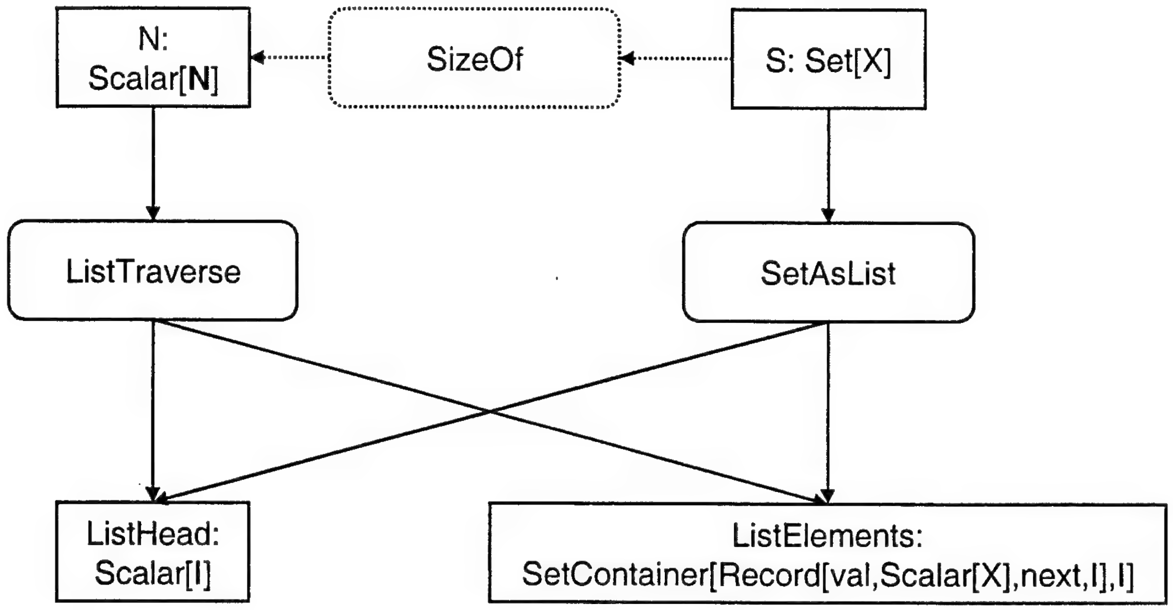


Figure 10. Indirect Implementation of SizeOf View

More generally, an indirect view implementation enforces a view specification between a view object O and source objects $O_1,\ldots,O_n$ by implementing each of these objects as direct views of some other objects $O'_1,\ldots,O'_m$.

Indirect views are closely related to the usual notion of implementation inheritance. With implementation inheritance, an object can have more than one type: a most specific type and one or more supertypes that are abstractions of this most specific type. The apparent state of the object, and the methods available to manipulate that state, depend on the type under which the object is being viewed. The state of the object under some supertype S is a function of the state of the object under the most specific type T. This function has also been called an abstraction function [LW94].

As stated earlier, two objects can be synchronized only if they have the same type. If an ojbect can have more than one type, the definition of synchronization becomes more difficult. We propose instead the following approach:

- Each object has exactly one type. The object's type defines its state space, its initial state, and how its state changes in response to requests, as described in Section 2.2.1.
- An object O of type T, when treated as an object of some supertype S through polymorphic assignment, is really a *different* object O'. Both are (direct) views of their implementation objects: $O = F_T(O_1,\cdots,O_n)$ and $O' = F_S(O_1,\cdots,O_n)$. O' is an indirect view of O; the indirect view function is the abstraction function. The polymorphic assignment generates a reference to O' from a reference to O.

In other words, *implementation inheritance is a specific form of indirect view implementation.*

2.4 Histories

A history is a record of an object's past and current states, with services for retrieving the changes between pairs of states. It is an application of the database log concept [GR93] to arbitrary objects. Examples of histories include a video clip (a history of an image), a sequence of timestamped sensor readings, an aircraft trajectory (history of position) and the output of a chemical process simulation. As with views, our goal is to have a very general definition of histories for object-oriented systems—one that allows various implementation approaches, including very efficient special-purpose implementations.

There are two common forms of history abstractions:

- *Value-based*: a sequence of (time, state *value*) pairs. This form of history provides services to determine an object's state as of specified time and (possibly) the sequence of object states recorded for a specified time interval. This form is common in process data loggers, which record timestamped sensor values. Most research in temporal databases also adopts a value-based model of history, in which a query is evaluated as of a specified time [TANS93].
- *Change-based*: a sequence of (time, state *change*) pairs. This form of history provides services to store and retrieve the sequence of changes to an object's state within a specified time interval. Database logs are of this form.

We adopt a change-based history abstraction, where the state changes are captured by the requests sent to the object. Gray and Reuter term this logical logging [GR93]. There are several reasons for this choice:

- Value-based history is a special case of change-based history, where each change is a complete replacement of the object's state. This is common for small objects of type $Scalar[X]$.
- For large objects, manipulating state changes is more efficient than manipulating states. It would be impractical to record the entire state of a database every time the database is updated.
- Many object types define no *implementation-independent* representation for values in their state domains. For example, the $Stack[X]$ object has $push(x)$ and pop requests for incrementally modifying the object's state, but no request for reading/writing the object's complete state. On the other hand, each object type T defines its possible state changes: the request domain Q_T.

Any object O can have a history. Section 2.4.1 defines a type $History[T,H,c]$ for any object type T. An object's *base history* records every change of state as it occurs in the object. Section 2.4.2 shows how O's base history can be updated by making it the source object of a view, where O is the view object. Section 2.4.3 shows how other histories can be defined as views of this base history. One purpose for doing this is to achieve varying levels of quality of service in object synchronization. Section 2.4.4 defines histories of views and discusses how they can be implemented.

2.4.1 *History Type Definition*

If O's type is T, then any history of O is of type $History[T,H,c]$, where H is a domain of *state identifiers* and $c: SQ_T^* \rightarrow Q_T^*$ is a compression function. State identifiers are used to label and select states in the history. There are many possible choices of domains for state identifiers, depending on the application. Integers, timestamps, multi-level version numbers, and transaction identifiers are common state identifiers. There must be a total order $<_H$ defined on H, so that the relative position of two states in an object's history can be determined by comparing their state identifiers. There must also be a *minimum* element $h_0 \in H$ and a distance function $d_H : H \times H \rightarrow \mathbf{R}$ to measure the difference between two state identifiers.

The type definition for $History[T,H,c]$ is as follows. At any time, the state of a history is of the form $(\langle(h_1,l_1),\ldots(h_n,l_n)\rangle,l)$, where $h_1,\ldots,h_n$ are state identifiers in strictly increasing order, and $l_1,\ldots,l_n$ and l are sequences of requests drawn from Q_T. The history represents a sequence of past states $s_0,\ldots,s_n$ and the current state s of an object of type T in the following way:

(10)
$$s_0 = \varepsilon_T$$
$$s_i = s_{i-1}!!_T\, l_i \text{ for } 1 \le i \le n$$
$$s = s_n!!_T\, l$$

In other words, the sequence of requests l_i, when applied to an object of type T, moves the object's state from s_{i-1} to s_i. Formally, the state space for type $History[T,H,c]$ is:

$$S = (H \times Q_T^*)^* \times Q_T^*$$

The initial state of a history has an empty sequence of past states, and the current state is the initial state of type T:

$$\varepsilon = (\langle\,\rangle,\langle\,\rangle)$$

The request and response domains are:

$$\begin{aligned} Q = &\{record(q') \mid q' \in Q_T\} \\ &\cup \{append(h) \mid h \in H\} \\ &\cup \{first\} \\ &\cup \{last\} \\ &\cup \{succ(h) \mid h \in H\} \\ &\cup \{get(h) \mid h \in H\} \\ &\cup \{getTail\} \end{aligned}$$

$$R = \{ok, badIdentifier, pastEnd\} \cup H \cup Q_T^*$$

Any request is possible in any state: $SQ = S \times Q$. Request $record(q')$ appends a request to the history, and compresses the sequence of requests recorded since the last identified state.

$$(\langle (h_1,l_1),\ldots,(h_n,l_n)\rangle,l)!record(q') = (\langle (h_1,l_1),\ldots,(h_n,l_n)\rangle, c(s_n, l \mid \langle q' \rangle))$$
$$(\langle (h_1,l_1),\ldots,(h_n,l_n)\rangle,l)?record(q') = ok$$

Here, s_n is shorthand for an expression over the history's state, as defined in Equation (10).

Any compression function c for type T can be used to process $record(q')$ requests, including the "null" compression function $c(s,l) = l$. If a stateless compression function is used, so that $c(s_n, l \mid \langle q' \rangle) = c'(l \mid \langle q' \rangle)$, the implementation can be simplified. The choice of compression function affects the history's state transitions, and therefore is a parameter of the type definition. Like all type definitions, $History[T,H,c]$ completely specifies the behavior of an object of this type. This is essential to Active Views mechanisms, as explained in Section 2.2.4.

Request $append(h)$ makes an identified state h from the current state of object O. The state identifier h must be greater than the state identifiers already in the history:

$$\begin{aligned}(\langle (h_1,l_1),\ldots,(h_n,l_n)\rangle,l)!append(h) = (&if\ h_n <_H h\\ &then\ (\langle (h_1,l_1),\ldots(h_n,l_n),(h,l)\rangle,\langle\,\rangle)\\ &else\ (\langle (h_1,l_1),\ldots(h_n,l_n)\rangle,l)\\ &)\end{aligned}$$
$$(\langle (h_1,l_1),\ldots,(h_n,l_n)\rangle,l)?append(h) = (if\ h_n <_H h\ then\ ok\ else\ badIdentifier)$$

Requests $first$, $last$, $succ(h)$, $get(h)$, and $getTail$ are for navigating the sequence of state identifiers and accessing the requests. None of these modify the state of the history. The responses to these requests are as follows:

$$(\langle (h_1,l_1),\ldots,(h_n,l_n)\rangle,l)?first = (if\ n > 0\ then\ h_1\ else\ pastEnd)$$
$$(\langle (h_1,l_1),\ldots,(h_n,l_n)\rangle,l)?last = (if\ n > 0\ then\ h_n\ else\ pastEnd)$$
$$(\langle (h_1,l_1),\ldots,(h_n,l_n)\rangle,l)?succ(h) = (if\ \exists k\ h = h_k \wedge 1 \le k < n\ then\ h_{k+1}\ else\ badIdentifier)$$
$$(\langle (h_1,l_1),\ldots,(h_n,l_n)\rangle,l)?get(h) = (if\ \exists k\ h = h_k \wedge 1 \le k \le n\ then\ l_n\ else\ badIdentifier)$$
$$(\langle (h_1,l_1),\ldots,(h_n,l_n)\rangle,l)?getTail = l$$

2.4.2 An Object as a View of its History

Suppose an object O of type T has a base history object BHO of type $History[T,H,c]$. BHO's state must always reflect O's current state. This can be accomplished by a view relationship $O = OHV[T,H](BHO)$, where $OHV[T,H]$ is the *object history view*. This view has parameters T and H, but we will omit these for brevity. The choice of compression function c does not affect the view definition.

This view is both forward and reverse updatable. Any request q sent to O causes a request $record(q)$ to be sent to BHO. Requests of form $append(h)$, $delete(h)$, $first$, $last$, $succ(h)$, $get(h)$, and $getTail$ are sent directly to BHO by other clients. As will be seen, none of these requests change the state of BHO in a way that changes the state of O.

The view function $F^{OHV} : (H \times Q_T^*)^* \times Q_T^* \to S_T$ is defined as follows:

$$F^{OHV}((\langle(h_1,l_1),\ldots,(h_n,l_n)\rangle,l)) = \varepsilon_T !!_T (l_1 \mid \ldots \mid l_n \mid l)$$

This just reflects the fact that the state of the object O is determined by the sequence of operations $\langle q_1,\ldots,q_m\rangle$ performed on it since its creation. The sequence $l_1 \mid \ldots \mid l_n \mid l$ is a compressed version of the original sequence, but is guaranteed to have result in the same state.

The source invariant SI^{OHV} and the view update constraint U^{OHV} are identically true; the view function F^{OHV} and the source and view types provide sufficient constraints on the view's behavior.

Figure 11 shows one way to implement the $OHV[T,H]$ view specification. Object O is implemented as a wrapper for two objects: O', another object of the same type, and the history object BHO. For each request q sent to O, the *HistoryWrapper* procedure does two things: (1) it sends the same request to the object O' and returns the response from O' to its client and (2) it sends request $record(q)$ to BHO. The implementation of object O' is a "real" implementation of type T. Note that the *HistoryWrapper* procedure implements two view specifications: $OHV[T,H]$ between view object O and source object BHO, and *Identity*$[T]$ (the identity function) between view object O and source object O'. If we assume that no clients except the *HistoryWrapper* procedure send a $record(q)$ request to BHO, then the state of O' will always equal the state of O that is predicted by the view relationship $O = OHV[T,H](BHO)$.

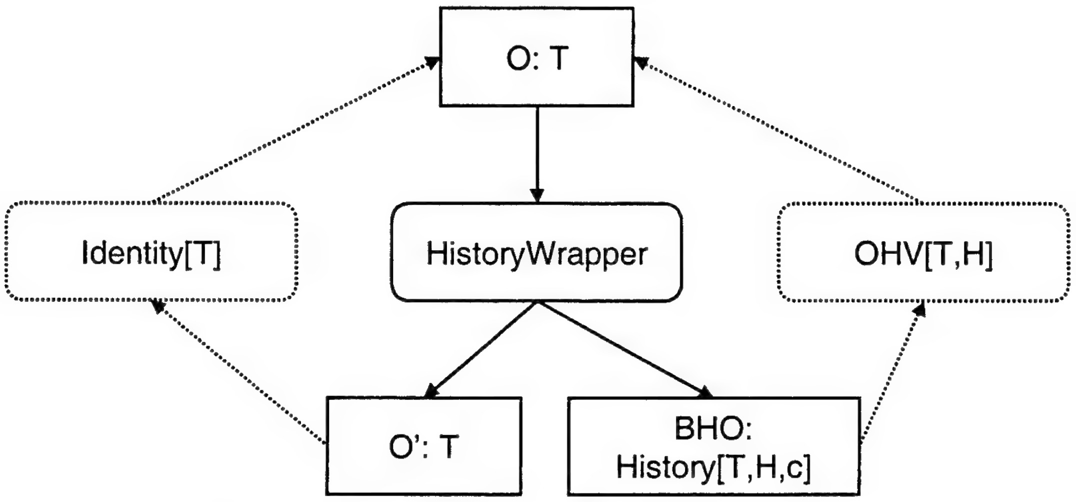


Figure 11. A Wrapper Implementation of a History

2.4.3 Views of Histories and Information Quality of Service

A history is an object, so views can be defined on it. One particular kind of view is useful for object synchronization: an *information quality of service* view. Synchronization works by sending the requests in a history of the source object to one or more target objects. In many cases, perfect synchronization is not required: the target objects' states can lag the source object by some amount of "time", where time is measured using state identifiers. Also, the target objects can differ somewhat from the source object's state, where the difference is measured using a distance function as defined in Section 2.2.6. In general, higher-fidelity synchronization requires more system resources; it should be possible to trade off synchronization fidelity and resource requirements.

The information quality of service view $IQoS[T,H,c,d_H,\Delta h,d_T,\Delta s]$ defines a view relationship $VHO = IQoS(SHO)$ between a view history object VHO and a source history object SHO, both of type $History[T,H,c]$. Besides the parameters T, H, and c taken from the source and view object types, the $IQoS$ view has four additional parameters: a *lag* Δh, a distance function d_H over state identifiers to measure lag, an *accuracy* Δs, and a distance function d_T over states to measure accuracy. VHO is derived from SHO by omitting those past states that are not sufficiently different (in terms of Δh and Δs) from earlier states that have been included in VHO.

F^{IQoS} has signature $F^{IQoS} : (H \times Q_T^*)^* \times Q_T^* \to (H \times Q_T^*)^* \times Q_T^*$. Any state of the source object SHO is of the form $ss = (\langle(sh_1,sl_1),\ldots,(sh_n,sl_n)\rangle, sl)$; the corresponding state $vs = F^{IQoS}(ss)$ is of the form $vs = (\langle(vh_1,vl_1),\ldots,(vh_m,vl_m)\rangle, vl)$. The state identifiers $vh_1,\ldots,vh_m$ in VHO are a subsequence of the state identifiers $sh_1,\ldots,sh_n$ in SHO. Exactly which identifiers are in this subsequence is determined by the *included* function: the value of $included(i)$ is true if sh_i is included in $vh_1,\ldots,vh_m$. The function $prev(i)$ returns the index within $sh_1,\ldots,sh_n$ of the last element prior to sh_i that is included in $vh_1,\ldots,vh_m$.

$$included(i) = \begin{cases} true & i = 0 \\ d_H(sh_{prev(i)}, sh_i) > \Delta h \vee d_T(ss_{prev(i)}, ss_i) > \Delta s & 1 \le i \le n \end{cases}$$

where $ss_i = \varepsilon_T !!_T (sl_1 \mid \ldots \mid sl_i)$

$$prev(i) = \max\{k \mid 0 \le k < i \wedge included(k)\}$$

From the *included* function, we can compute the length m of the sequence $vh_1,\ldots,vh_m$ and, for any index j in that sequence, the index $index(j)$ of the same element within $sh_1,\ldots,sh_n$.

$$m = \left|\{i \mid 1 \le i \le n \wedge included(i)\}\right|$$

$$index(j) = \begin{cases} 0 & j = 0 \\ \min\{i \mid i > index(j-1) \wedge included(i)\} & 1 \le j \le m \end{cases}$$

Finally, we can construct the view's state $vs = (\langle(vh_1,vl_1),\ldots,(vh_m,vl_m)\rangle, vl)$. The sequence of requests vl_j is generated by compressing the sequences of requests that follow the previous state identifier $sh_{prev(index(j))}$ in SHO.

(11) $vh_j = sh_{index(j)} \quad 1 \le j \le m$

$vl_j = c(ss_{prev(index(j))}, (sl_{prev(index(j))+1} | \ldots | sl_{index(j)})) \quad 1 \le j \le m$

$vl = c(ss_{index(m)}, (sl_{index(m)+1} | \ldots | sl_n | sl))$

Depending on the values chosen for Δh and Δs, this compression can lead to a considerable reduction in the number of requests in *VHO* compared to *SHO*, and hence reduced synchronization resource requirements.

This view specification can be implemented in a number of ways. Figure 12 shows one way. It requires the compression function c to support the distance function d_H, i.e. there must be a function $d'_T : Q^*_T \to \mathbf{R}$ such that $\forall (s,l) \in SQ^*_T \; d_T(s, s!!_T \, l) = d'_T(c(s,l))$. (See Section 2.2.7.) The technique used here is similar to the one shown in Figure 11 to implement a history: a wrapper procedure *IQoSWrapper* receives requests to a source history object *SHO*, and forwards them to a "real" implementation *SHO′* of the base history object. It also forwards the requests to a separate history object *VHO*, except that before forwarding an *append*(*h*) request, it sends a *last* request and a *getTail* request to *VHO* to obtain vh_m and vl from the state of the view. It computes $d_H(vh_m, h)$ and $d_T(vs_m, vs_m!!vl)$, which equals $d'_T(vl)$ since vl is already compressed. If either distance exceeds its respective bound Δh or Δs, the *IQoSWrapper* procedure sends the *append*(*h*) request to *VHO*; otherwise it does not. The result is that the state of *VHO* includes a subset of the state identifiers in *SHO*, and the same requests as in *SHO*, except they are further compressed.

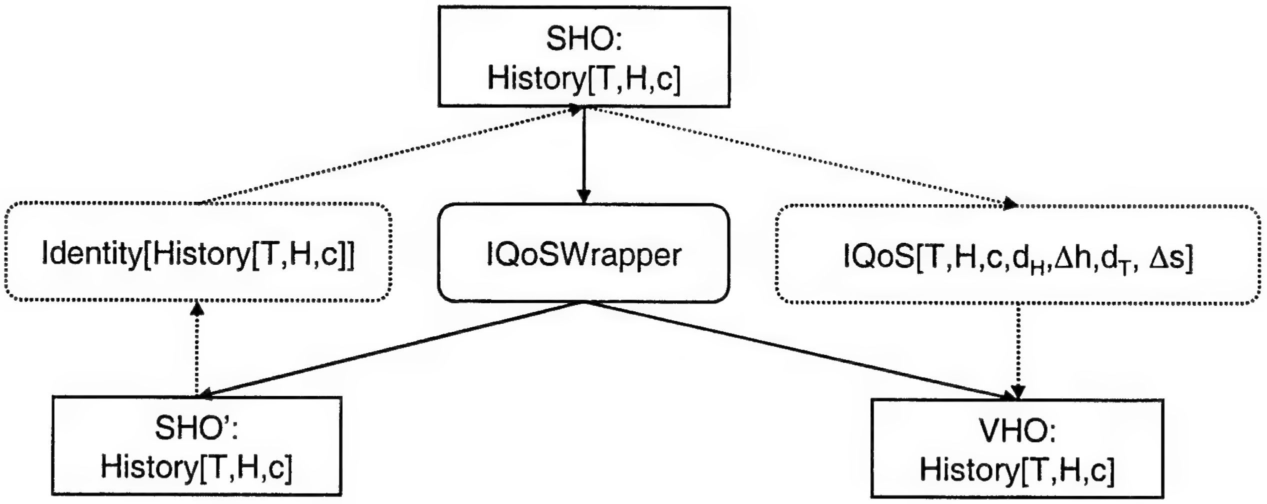


Figure 12. An Implementation of the IQoS View Specification

Other kinds of view of histories are possible. For instance, process historians are specialized data servers that record timestamped sequences of scalar values, typically the outputs of sensors that measure temperature, pressure, flow, etc. These historians can provide clients applications both raw sequences of recorded values and sequences generated by interpolation, filtering, and other functions applied to the raw sequences. Such computed sequences are really views of the underlying raw sequences.

2.4.4 Histories of Views

Since a view is an object, a view can have a history, too. In an active information system, a history of a view is needed when a client application wants to be informed (via object synchronization) of changes in the state of the view rather than of the underlying source objects. The view function may perform significant information abstraction or transformation to meet the client application's needs.

Figure 13 shows the view relationships among a view object O, its source objects $O_1,\ldots,O_n$, and their respective histories HO and $HO_1,\ldots,HO_n$. The relationship between O and $O_1,\ldots,O_n$ is defined by an arbitrary view specification V. The relationship $OHV[T,H]$ between O and HO, and the similar relationships between the source objects and their histories, were defined earlier. The relationship between HO and $HO_1,\ldots,HO_n$ is defined by a view specification VHV. This relationship requires that each of the histories use the same state identifier domain H.

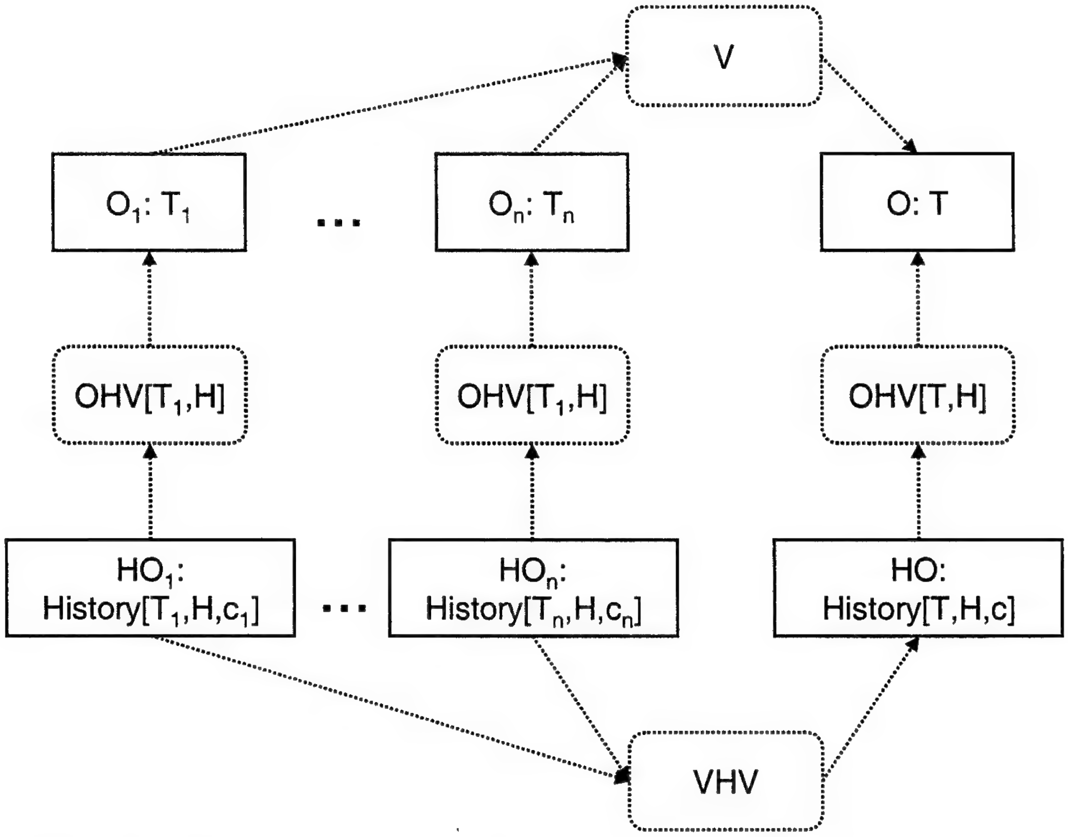


Figure 13. The View Relationships Among a View Object, Its Source Objects, and Their Histories

The view VHV is specified as follows. There is a non-trivial source invariant I^{VHV}: the sequence of state identifiers that appear in each source object's history must always be the same. In other words, the states $sho_1,\ldots,sho_n$ of the source histories $HO_1,\ldots,HO_n$ can be written:

(12) $sho_i = (\langle(h_1,l_1^{(i)}),\ldots,(h_m,l_m^{(i)})\rangle, l^{(i)})$

This requirement is imposed because the past states of the view object O, as represented in its history HO, can only be determined for those state identifiers that appear in every source object's history. (If the available histories of the source objects don't have identical sequences of state identifiers, it may be possible to construct views of those histories, as discussed earlier, to meet this requirement. The views could omit state identifiers, interpolate between recorded states, etc., to meet the requirement.)

Figure 13 is somewhat misleading in that it implies that there is a single view specification VHV —in particular a single view function F^{VHV} —that is implied by the other relationships among the various objects. In fact, there can be more than one definition of F^{VHV} that is consistent with the other relationships. The flexibility in the definition of F^{VHV} arises from the lack of information about the ordering of requests sent to the different source objects $O_1,\ldots,O_n$ and recorded in their histories $HO_1,\ldots,HO_n$. The common sequence of state identifiers imposes some degree of order: if state identifiers h_p and h_q appear in both sho_i and sho_j (the states of history objects HO_i and HO_j) and if $h_p <_H h_q$, then the sequences of requests $l_p^{(i)}$ and $l_p^{(j)}$ were sent to O_i and O_j, respectively, strictly before the sequences of requests $l_q^{(i)}$ and $l_q^{(j)}$ were sent to those objects. However, we don't know how requests in $l_p^{(i)}$ were interleaved in time with requests in $l_p^{(j)}$. This lack of time ordering information is typical of asynchronous computing systems [BM93]. Therefore we don't know the sequence of state transitions that the view object O underwent between state identifiers h_{p-1} and h_p. If the compression function c produces the *same* result for any of the possible sequences of state transitions in the view object O, the value of l_p is completely determined; otherwise it is not. Therefore we will present constraints on the definition of F^{VHV} rather than a complete definition.

The view function F^{VHV} has the following signature:

$$F^{VHV} : ((H \times Q_{T_1}^*)^* \times Q_{T_1}^*) \times \ldots \times ((H \times Q_{T_n}^*)^* \times Q_{T_n}^*) \rightarrow ((H \times Q_T^*)^* \times Q_T^*)$$

The sequence of state identifiers in HO must be the same as those in the source objects' histories. So, if the states $sho_1,\ldots,sho_n$ of the source histories $HO_1,\ldots,HO_n$ are as written in Equation (12), then the state sho of the view history HO can be written as:

$$sho = (\langle (h_1, l_1), \ldots, (h_m, l_m) \rangle, l)$$

Finally, the sequences of requests in sho must guarantee that the states of the source and view histories obey the view function F^V at each recorded point in history:

(13) $$\varepsilon_T !!_T (l_1 | \ldots | l_p) = F^V(\varepsilon_{T_1} !!_{T_1} (l_1^{(1)} | \ldots | l_p^{(1)}), \ldots, \varepsilon_{T_n} !!_{T_n} (l_1^{(n)} | \ldots | l_p^{(n)})) \quad 1 \le p \le m$$
$$\varepsilon_T !!_T (l_1 | \ldots | l_p | l) = F^V(\varepsilon_{T_1} !!_{T_1} (l_1^{(1)} | \ldots | l_m^{(1)} | l^{(1)}), \ldots, \varepsilon_{T_n} !!_{T_n} (l_1^{(n)} | \ldots | l_m^{(n)} | l^{(n)}))$$

If a request is sent directly to HO, equations (12) and (13) and the type definitions $T_1,\ldots,T_n$ impose significant constraints on the behavior of the view in updating the source histories $HO_1,\ldots,HO_n$. No further source update constraint is necessary—U^{VHV} is identically true.

There are many ways of implementing a history of a view. The approach chosen depends on specific properties of the source and object types and how the view V is implemented. Figure 14 shows a fairly general approach that assumes that view specification V is implemented as a virtual view, i.e. view object O's state is computed from the states of source objects $O_1,\ldots,O_n$. The number of objects and procedures may appear daunting, but only the shaded objects are "real", materialized objects; the rest are views of those objects implemented through the procedures shown in the figure. Also, many of the procedures are fairly trivial.

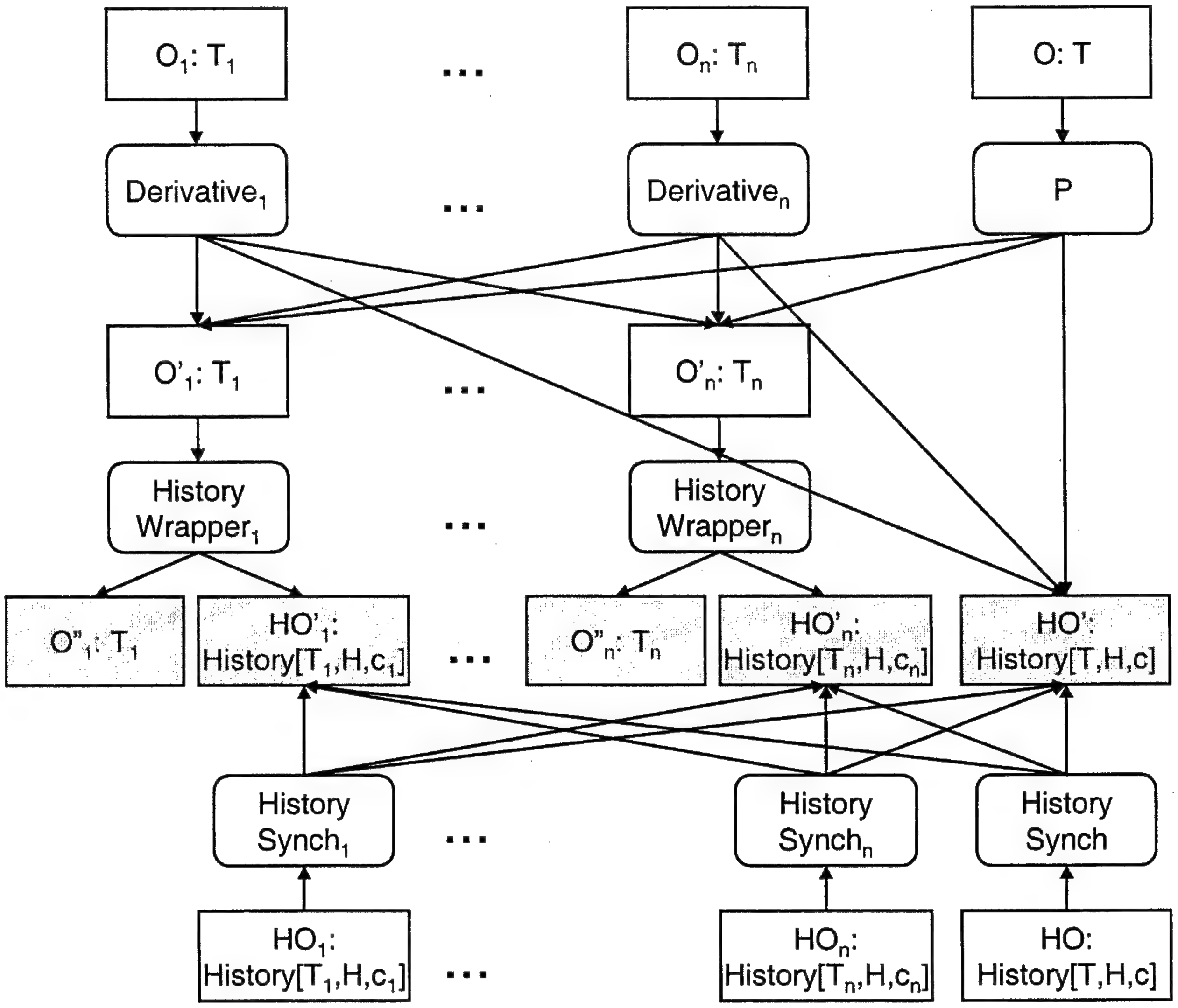


Figure 14. An Implementation of a History of a View

Starting from the top and bottom of the figure, objects $O_1,\ldots,O_n$, $HO_1,\ldots,HO_n$, O, and HO are the source objects, their histories, the view object, and its history that are presented to external clients. Procedures $Derivative_1,\ldots,Derivative_n$ and P are the keys to the view implementation.

The purpose of $Derivative_i$ is to intercept a request q sent to O_i and to compute the requests that must be added to H (via *record* requests) to reflect the corresponding change in state of the view object O. It can access, but not modify, the states of the other source objects in computing this change. The procedure P performs a similar function by intercepting requests sent to O, computing the corresponding requests that must be sent to the source objects, and recording the request in HO. The purpose of $HistoryWrapper_i$ is to intercept requests from those procedures to the source objects, to send them to the "real" implementations of the source objects, and to record them in the "real" implementations of the source histories. The purpose of the *HistorySynch* procedures is to guarantee that the source and view histories have the same sequence of state identifiers. Whenever an external client sends an $append(h)$ request to one of these histories, the *HistorySynch* procedure intercepts the request and forwards it to all of the histories.

Simpler implementations of a history of a view are possible for specific source object types and views. For instance, consider a source object SO of type $Record[f_1,T_1,\ldots,f_n,T_n]$ and a view object VO of type T_i, where $VO = Project_i[SO]$. The view function $Project_i$ maps the state $(s_1,\ldots,s_n)$ of SO to state s_i of VO. A history of VO can be implemented as a "virtual" view of the history of SO, rather than as a materialized view as in Figure 14

2.5 Object Synchronization

Object synchronization is the final piece in the Active Views puzzle, and with all that has come before it, it is easy to define. The purpose of object synchronization is to propagate state changes from a source object to one or more target objects of the same type, so that their states are synchronized. Object synchronization is similar to the usual notion of event services such as those defined for CORBA [OMG], COM [CHAP96], and Java [FLAN97], but here we ascribe a more specific meaning to the concept of event: *an event is a complete description of a change of state in a specific, identified object.* The state changes that an object can undergo are the requests defined by the object's type. The CORBA, COM, and Java event services allow various types of events to be defined. These event types can have parameters to carry instance-specific information. However, there is no requirement that an event identify a specific object as its source, or that the event be a complete description of a state change.

A commonly-used event-oriented pattern is for a source object to announce to subscribers . (perhaps by a method invocation) *that* its state has changed, but not exactly *how*. Each subscriber must then query the source object to determine the new state. Gamma et al. [GHJV95] call this the *observer pattern*. Object synchronization is a different, more efficient way of achieving the same end: the event carries the information about *how* the state changed, so that the subscriber need not query the source further. In many cases, the event encodes the *difference* between the source's old and new states. The recipient may use this information for various purposes, e.g. to highlight the changes on a display. In the observer pattern, the subscriber must explicitly compare the source's old state (which it stores locally) to the new state it gets from the source.

Object synchronization differs from the usual notion of event services in another way. Event services allow a client application to subscribe to a stream of events that emanate from some object. The subscriber receives those events that occur *after* the subscription starts. Even if the events carry a complete description of the source object's state change, this information is insufficient to allow the recipient to track the source object's state, since the recipient doesn't know the state of the source object at the time the subscription took effect. Object synchronization solves this problem by synchronizing the recipient's state to the current state of the source object before sending it new events.

Object synchronization uses a history of the source object as its source of state changes. A synchronizer procedure (Figure 15) monitors the history for new state identifiers and applies the corresponding sequences of requests to the target objects. It also maintains a record of the synchronization status of the target objects. This synchronization can occur either automatically or at the explicit request of a client application. As stated earlier, the source and target objects must be of the same type. Since our type definitions are complete—they completely define the behavior of the object in response to requests—the target objects are guaranteed to have the same state as the source object.

The synchronizer procedure is an implementation of the *Synchronizer* view specification. The view object is an object of type *SynchronizationControl* that, as the name suggests, provides requests for controlling the synchronization. Unlike most of the views specified so far, *Synchronizer* is more notable for the state changes that occur in the view's source objects when a request sent to the view object, than for the state of the view object itself. The *SynchronizationControl* type definition and the *Synchronizer* view specification are given below.

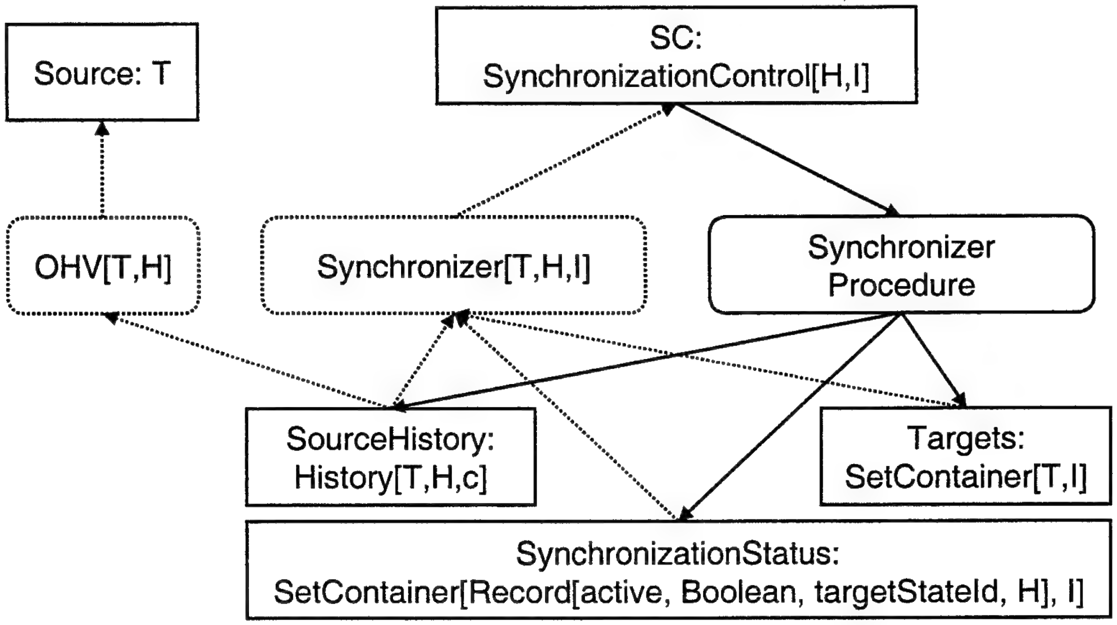

Source: T
SC: SynchronizationControl[H,I]
OHV[T,H]
Synchronizer[T,H,I]
Synchronizer Procedure
SourceHistory: History[T,H,c]
Targets: SetContainer[T,I]
SynchronizationStatus: SetContainer[Record[active, Boolean, targetStateId, H], I]

Figure 15. Object Synchronization Objects, View Specification, and View Implementation

Figure 16 shows an example of object synchronization: aperiodic updates to a tabular display window based on changes to a database table. Both the database table and the window are treated as objects of type *Table* that accept insert, update and delete operations. Therefore the updates that database clients apply to the database table can be directly applied to the display to maintain synchronization.

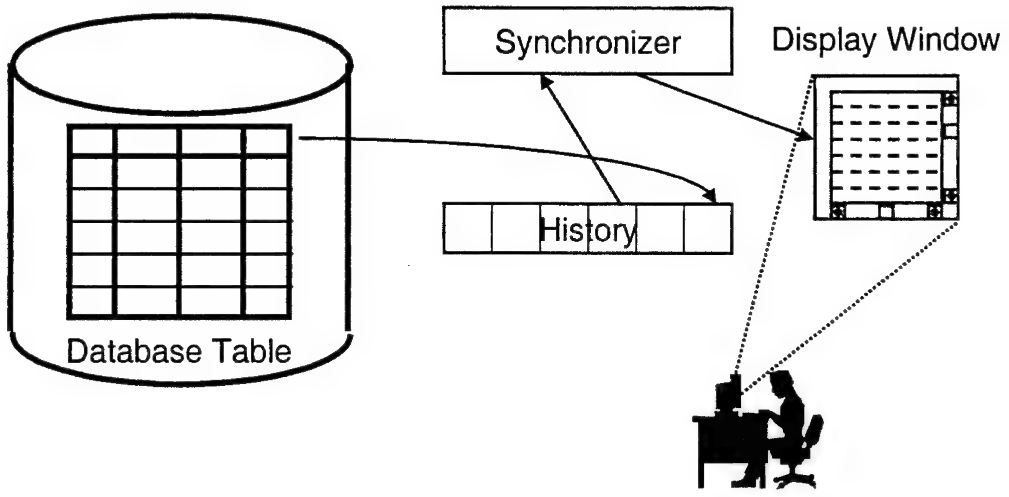


Figure 16. Example of Object Synchronization

2.5.1 SynchronizationControl Type Definition

The $SynchronizationControl[H, I]$ type has two parameters: H, the domain of state identifiers used in the source object's history, and I, the domain of object identifiers used by the container object in which all synchronization target objects are contained.

The state domain S of type $SynchronizationControl[H, I]$ records the sequence of state identifiers in the source object's history, and for each target object, a boolean value that is true if synchronization is currently active for that object (otherwise it is paused) and a state identifier indicating the current state of the object. Initially, the source object's history is empty and there are no target objects:

$$
\begin{aligned}
&S = H^* \times (I \nrightarrow (Boolean \times H)) \\
&\varepsilon = (\langle\,\rangle, \{\,\})
\end{aligned}
$$

The $SynchronizationControl[H, I]$ type provides several forms of request. They are all permissible in any state of the object.

$$
\begin{aligned}
Q = &\{append(h) \mid h \in H\} \\
&\cup \{insert(i) \mid i \in I\} \\
&\cup \{delete(i) \mid i \in I\} \\
&\cup \{synch(i) \mid i \in I\} \\
&\cup \{pause(i) \mid i \in I\} \\
&\cup \{synchToState(i, h) \mid i \in I \wedge h \in H\} \\
&\cup \{tick\} \\
SQ = &\, S \times Q \\
R = &\, \{ok, badObjectIdentifier, badStateIdentifier\}
\end{aligned}
$$

The $append(h)$ request adds a new state identifier to the history. A client application can send this request to an object of type $SynchronizationControl$, or directly to the underlying history object.

$$
\begin{aligned}
(\langle h_1,\ldots,h_n\rangle, synchStatus)!append(h) = (&if\ h_n <_H h \\
&then\ (\langle h_1,\ldots,h_n,h\rangle, synchStatus) \\
&else\ (\langle h_1,\ldots,h_n\rangle, synchStatus) \\
&)
\end{aligned}
$$

$$(\langle h_1,\ldots,h_n\rangle, synchStatus)?append(h) = (if\ h_n <_H h\ then\ ok\ else\ badStateIdentifier)$$

The $insert(i)$ request adds an object, identified by object identifier i, to the set of synchronization target objects. The object is assumed to be in the initial state for its type, which must be the same type as the synchronization source object.

$$
\begin{aligned}
(\langle h_1,\ldots,h_n\rangle, synchStatus)!insert(i) = (&if\ i \in \mathrm{dom}(synchStatus) \\
&then\ (\langle h_1,\ldots,h_n\rangle, synchStatus) \\
&else\ (\langle h_1,\ldots,h_n\rangle, synchStatus \cup \{(i,(false,h_0))\}) \\
&)
\end{aligned}
$$

$$
\begin{aligned}
(\langle h_1,\ldots,h_n\rangle, synchStatus)?insert(i) = (&if\ i \in \mathrm{dom}(synchStatus) \\
&then\ badObjectIdentifier \\
&else\ ok \\
&)
\end{aligned}
$$

The $delete(i)$ request removes an object from this set:

$$
\begin{aligned}
(\langle h_1,\ldots,h_n\rangle, synchStatus)!delete(i) = (&if\ i \in \mathrm{dom}(synchStatus) \\
&then\ (\langle h_1,\ldots,h_n\rangle, synchStatus - \{(i, synchStatus(i))\}) \\
&else\ (\langle h_1,\ldots,h_n\rangle, synchStatus) \\
&)
\end{aligned}
$$

$$(\langle h_1,\ldots,h_n\rangle, synchStatus)?delete(i) = ok$$

The $synch(i)$ request marks synchronization *active* for target object i, and brings the target object to the last recorded state of the source object:

$$
\begin{aligned}
(\langle h_1,\ldots,h_n\rangle, synchStatus)!synch(i) = (&if\ i \in \mathrm{dom}(synchStatus) \\
&then\ (\langle h_1,\ldots,h_n\rangle, synchStatus \uplus \{(i,(true,h_n))\}) \\
&else\ (\langle h_1,\ldots,h_n\rangle, synchStatus) \\
&)
\end{aligned}
$$

$$
\begin{aligned}
(\langle h_1,\ldots,h_n\rangle, synchStatus)?synch(i) = (&if\ i \in \mathrm{dom}(synchStatus) \\
&then\ ok \\
&else\ badObjectIdentifier \\
&)
\end{aligned}
$$

The $pause(i)$ request marks synchronization *inactive* for target object i, leaving the object in its current state:

$$\begin{aligned}(\langle h_1,\ldots,h_n\rangle, \mathit{synchStatus})!\mathit{pause}(i) = (&\mathit{if}\ i \in \mathrm{dom}(\mathit{synchStatus})\\ &\mathit{then}\ (\langle h_1,\ldots,h_n\rangle, \mathit{synchStatus} \uplus \{(i,(\mathit{false},h'))\})\\ &\mathit{else}\ (\langle h_1,\ldots,h_n\rangle, \mathit{synchStatus})\\ &)\\ &\text{where } (a,h') = \mathit{synchStatus}(i)\end{aligned}$$

$$\begin{aligned}(\langle h_1,\ldots,h_n\rangle, \mathit{synchStatus})?\mathit{pause}(i) = (&\mathit{if}\ i \in \mathrm{dom}(\mathit{synchStatus})\\ &\mathit{then\ ok}\\ &\mathit{else\ badObjectIdentifier}\\ &)\end{aligned}$$

The $\mathit{synchToState}(i,h)$ request advances the state of target object i to the source object state identified by h in the source's history, then marks synchronization inactive for this target object:

$$\begin{aligned}(\langle h_1,\ldots,h_n\rangle, \mathit{synchStatus})!\mathit{synchToState}(i,h) = (&\mathit{if}\ i \notin \mathrm{dom}(\mathit{synchStatus})\\ &\mathit{then}\ (\langle h_1,\ldots,h_n\rangle, \mathit{synchStatus})\\ &\mathit{else\ if}\ h \notin \langle h_1,\ldots,h_n\rangle \vee h <_H h'\\ &\mathit{then}\ (\langle h_1,\ldots,h_n\rangle, \mathit{synchStatus})\\ &\mathit{else}\\ &(\langle h_1,\ldots,h_n\rangle, \mathit{synchStatus} \uplus \{(i,(\mathit{false},h))\})\\ &)\\ &\text{where } (a,h') = \mathit{synchStatus}(i)\end{aligned}$$

$$\begin{aligned}(\langle h_1,\ldots,h_n\rangle, \mathit{synchStatus})?\mathit{synchToState}(i,h) = (&\mathit{if}\ i \notin \mathrm{dom}(\mathit{synchStatus})\\ &\mathit{then\ badObjectIdentifier}\\ &\mathit{else\ if}\ h \notin \langle h_1,\ldots,h_n\rangle \vee h <_H h'\\ &\mathit{then\ badStateIdentifier}\\ &\mathit{else\ ok}\\ &)\\ &\text{where } (a,h') = \mathit{synchStatus}(i)\end{aligned}$$

Finally, the *tick* request advances the state of all target objects for which synchronization is active to the last recorded state of the source object. The *tick* request is so named because in many implementations, a clock-driven client application will send this request periodically to the *SynchronizerControl* object to activate synchronization.

$$\begin{aligned}&(\langle h_1,\ldots,h_n\rangle, \mathit{synchStatus})!\mathit{tick}\\ &\quad = (\langle h_1,\ldots,h_n\rangle, \mathit{synchStatus} \uplus \{(i,(\mathit{true},h_n)) \mid \exists h'\ (i,(\mathit{true},h')) \in \mathit{synchStatus}\})\\ &(\langle h_1,\ldots,h_n\rangle, \mathit{synchStatus})?\mathit{tick} = \mathit{ok}\end{aligned}$$

2.5.2 Synchronizer View Specification

Here is a specification of the $\mathit{Synchronizer}[T,H,I]$ view. The parameters of the view specification are the source object type, the domain for state identifiers in the source object's history, and the domain for object identifiers in the set of target objects.

As shown in Figure 15, the view has three source objects: a history object, a synchronization status object, and an object that contains the synchronization targets. The synchronization status object consists of a record for each target object; the record has a field that records whether synchronization is currently active for that object and a field that records the state identifier that represents the current state of the target object:

$$T_1 = History[T, H, c]$$
$$T_2 = SetContainer[Record[active, Boolean, targetStateId, H], I]$$
$$T_3 = SetContainer[T, I]$$

The view object is of type $SynchronizationControl[H, I]$ as defined above.

$$T = SynchronizationControl[H, I]$$

Based on these types, the view function $F^{Synchronizer[T,H,I]}$ has the following signature:

$$\begin{aligned}F^{Synchronizer[T,H,I]} &: ((H \times Q_T^*)^* \times Q_T^*) \times (I \nrightarrow (Boolean, H)) \times (I \nrightarrow S_T) \\ &\rightarrow H^* \times (I \nrightarrow (Boolean \times H))\end{aligned}$$

The view function computes, from the states of the source objects, the sequence of state identifiers in the source object's history and the synchronization status of each target object. The computation of the synchronization status allows the possibility that a target object is listed in the synchronization status object but has been deleted from the target container, or that the target object has been inserted in the target container but is not yet in the synchronization status object. In the latter case, the target object is assumed to be in the initial state for type T and that synchronization is paused.

$$\begin{aligned}&F^{Synchronizer[T,H,I]}((\langle (h_1, l_1), \ldots, (h_n, l_n) \rangle, l), synchStatus, targetState) \\ &\quad = (\langle h_1, \ldots, h_n \rangle, \\ &\qquad \{(i, synchStatus(i)) \mid i \in \mathrm{dom}(targetState) \cap \mathrm{dom}(synchStatus)\} \\ &\qquad\quad \cup \{(i, (false, h_0)) \mid i \in \mathrm{dom}(targetState) - \mathrm{dom}(synchStatus)\} \\ &\qquad)\end{aligned}$$

The source invariant requires that the state of each target object be an identified state of the source object, where the state identifier is the one found in the synchronization status object.

$$\begin{aligned}&I((\langle (h_1, l_1), \ldots, (h_n, l_n) \rangle, l), synchStatus, targetState) \Leftrightarrow \forall i \in \mathrm{dom}(targetState) \\ &\quad targetState(i) = (if\ i \in \mathrm{dom}(synchStatus)\ then\ \varepsilon_T !!_T\ (l_1 \mid \ldots \mid l_j)\ else\ \varepsilon_T) \\ &\quad \text{where } (a, h_j) = synchStatus(i)\end{aligned}$$

The update constraint $U^{Synchronizer[T,H,I]}$ is identically true; there is no need for further constraint on the behavior of the view.

2.5.3 Synchronizer Procedure Implementation

Figure 15 shows a direct implementation of the $Synchronizer[T, H, I]$ view by a *Syncrhonizer* procedure. This procedure works as follows in response to requests sent to the *SC* (*SynchronizationController*) object.

- In response to an *append*(h) request, the procedure sends an *append*(h) request to *SourceHistory*.
- In response to an *insert*(i) request, the procedure sends a *member*(i) request to *Targets* to determine whether i references an existing target. If so, the procedure returns a *badObjectIdentifier* response. Otherwise, it sends *insert*(i) requests to *SynchronizationStatus* and *Targets*.
- In response to a *delete*(i) request, the procedure sends *delete*(i) requests to *SynchronizationStatus* and *Targets*.
- In response to a *synch*(i) request, the procedure sends a *member*(i) request to *Targets* to determine whether i references an existing target. If not, the procedure returns a *badObjectIdentifier* response. Otherwise, it sends a *member*(i) request to *SynchronizationStatus* to determine whether an entry for that target object has been created. If not, it creates one using *insert*(i). In any case, it sends an $invoke(i, invoke\_active(set(true)))$ message to *SynchronizationStatus* to turn on synchronization for this target object. It sends an $invoke(i, invoke\_targetStateId(get))$ request to determine the current state of the target object. It then sends *last*, *get*(h), and *succ*(h) requests to *SourceHistory* to retrieve the requests it will send to the target object.

The logic for *pause*(i) and *synchToState*(i) is similar. The *tick* request poses a small problem: implementing it requires a way to get the identifiers of all objects contained in *Targets*, so that their states can be synchronized with the source object. The *SetContainer* type must be extended to support this functionality. If this is done, then implementing the *tick* request is similar to implementing *synch*(i).

2.5.4 Discussion

Object synchronization provides two degrees of freedom in defining the sequence of events that a target object receives. The first is the type of the source object. If no source object is available that generates the desired types of events, one can often implement a view object whose type generates those events, where the view function provides the desired information selection, abstraction, or transformation. The fact that a view can have a history, and hence that object synchronization can use a view as a source object, provides significant application flexibility. For instance:

- Suppose we want to synchronize two relational databases that have identical SQL schemas. We can synchronize the databases at the physical (implementation) level, so that the file structures, indexes, etc. are identical. Alternatively, we can synchronize the databases at the logical (abstract) level, so that the tables have identical contents but have different underlying physical structures optimized to support different workloads. The logical level is a view of the physical level.
- Consider an object of type *Set*$[I]$ whose state contains references to objects of type T. If this object is used as a synchronization source object, targets learn of identifiers inserted

into and deleted from the set. Alternatively, we can use the *SubsetContainer* view to create an object of type *SetContainer*$[T, I]$ as the synchronization source, so that targets also learn about changes in the states of the referenced objects. Both alternatives are useful.

- A video camera can be modeled as an object whose state is an array of pixels, where the state is completely rewritten several times per second. Views of this object can be constructed with feature extraction and object recognition as the view functions, so that the camera's state can be abstracted to a set of features or a set of recognized objects. Any of these objects—the pixel array, the set of features, or the set of recognized objects—can serve as a synchronization source, depending on application needs.

The second degree of freedom is information quality of service. The frequency and magnitude of state changes reported to target objects can be adjusted by using an IQoS view of the source object's history with appropriate selections for the various parameters. (See Section 2.4.3.) Quality of service can be adjusted to accommodate low bandwidth communications media or other computing and communication resource constraints.

2.6 Relationship to Active Databases

The relationship between individual Active Views mechanisms and previous work has been discussed in previous sections. Here we compare Active Views to the usual notion of active databases as an overall approach to active information systems.

Traditionally, databases have been passive repositories that cannot alert a user to database changes. These changes could be recognized only by issuing the same query repeatedly. This is obviously inefficient. Active database management systems, as defined in the database research literature, extend conventional relational or object-oriented DBMSs through the addition of *event-condition-action* (ECA) rules [DGG95]. Numerous active DBMS research prototypes have been developed [WC96]. A basic form of ECA rules, known as triggers, is implemented in many commercial relational DBMSs.

The features and semantics of ECA rules vary among DBMSs [PATE93]. However, the following is representative. An ECA rule has three parts: a specification of the event required to trigger the rule, a condition to be evaluated when the rule is triggered, and an action to perform if the condition is true. The action may be a further database update, a transaction abort, or a signal to some application.

An event represents a change of the state of the DBMS or other occurrence of significance to the application. Each event belongs to some *event class*. Active object-oriented DBMSs typically associate event classes with the operations defined for a given object class, so that an event occurs when the operation is invoked on a particular object. The event may be defined to occur before or after the operation is executed. Active relational DBMSs typically provide a fixed set of event classes for each table, corresponding to insert, delete, and update operations.

ECA rules have several applications, including:

- Enforcing semantic integrity constraints—Rules can detect when a database is updated, check whether an integrity constraint is violated, and either abort the transaction or update the database further to restore semantic integrity. For example, on an insert operation, a rule's condition can check for existence of another row with the same key, and the action can abort the transaction if one exists. Similarly, rules can be used to verify foreign key constraints and to implement cascade update and delete actions.

- Maintaining replicated data and materialized views—When there are multiple copies of a database table or other object in a distributed system, rules can be used to detect updates to one copy and either update other copies immediately or log the update for subsequent propagation. Materialized views, either local or remote to the source database, can be maintained in a similar fashion.
- Propagating database updates to applications—The application can do any number of things with this information, e.g. update a display, send a mail message, or perform further database updates based on application-specific computation. Such application processing is beyond the visibility and control of the DBMS.

ECA rules, as they have been defined for active databases, are a powerful extension to conventional DBMSs. However, ECA rules have two major drawbacks when compared to Active View mechanisms. First, they are *nondeterministic*—the effect of a set of rules can differ depending on which of several rules fires first. As a result, the state of the database can take any of a number of diverging trajectories. Active DBMSs typically provide a conflict resolution mechanism such as rule precedence or priority to reduce or eliminate such nondeterminism. However, these mechanisms add to the complexity of rule execution semantics. In contrast, the behavior of views, histories, and object synchronization are deterministic, with a single exception: the state of a history of a view may be is incompletely determined by the states of the histories of the view's source objects. This nondeterminism arises from the lack of a total order of events in a distributed system. However, the nondeterminism is limited in the sense that the history of the view must be consistent with the history of the source objects for all recorded state identifiers (see Equation (13).)

Second, ECA rules are *not composable*—No way has been defined to compose sets of rules into ever-larger rule bases and to reason about that composition. The experience with forward-chaining rules for expert systems is that they become brittle and intellectually unmanageable as the number of rules increases. In contrast, views have clean composition semantics based on mathematical composition of view functions.

Moreover, the applications of ECA rules listed above can also be accomplished with Active Views. A semantic integrity constraint is a predicate (boolean-valued function) over the state of a database or an element of it. The predicate can be used to define a view object whose state is *true* if the integrity constraint holds. The view's state can be updated incrementally based on changes to the underlying database; if the view's state is false at the end of the transaction, the DBMS can abort the transaction. This is a recognized application of materialized views [GM95]. Active Views obviously also support data replication and application notification.

There has been research into how *events* (as opposed to rules) can be composed into higher-level events. A good example of this is Snoop [CKAK94]. A composite event occurs when more primitive events occur in a particular pattern over time, e.g. a database table update followed by the clock reaching 12:00. Snoop provides several operators (*and*, *or*, *sequence*, etc.) for composing events.

Active Views can accomplish the equivalent of event composition very simply. For example, suppose there is a set of primitive events $\{a, b, c, \ldots\}$ and that a composite event is defined using a regular expression over the set of primitive events. The composite event occurs whenever a sequence of primitive events occurs that matches the regular expression. It is well known that for any regular expression there is a finite state automaton that recognizes sequences of symbols (the events) defined by the regular expression. (See, for example, [AU72]). A finite state automaton is defined by a set of states, a set of input symbols (events), a state transition function, an initial state, and a set of final states. One can easily define a corresponding object type in which the request domain is the set of primitive events. The type's response function would return *true* whenever the object entered a final state (indicating that the composite event had occurred), otherwise *false*. More complex event composition operators, such as those that generate composite events based on parameter values carried by the primitive events, can also be mapped to object types.

Finally, we believe that reasoning about states is easier than reasoning about sequences of events. In object-oriented analysis and design, reasoning is in terms of states, invariants, and pre- and post-conditions [MEYE97,KR94]. Furthermore, reasoning in terms of states allows us to employ well-understood notions such as function composition and distance functions.

2.7 Future Work

There are a number of ways the current work can be extended. First, of course, is to develop more substantial implementations of Active View concepts in a specific application domain. A number of application domains involving monitoring and control of large-scale dynamic systems can use Active Views to good benefit. Examples include:

- An information dissemination service to which client applications could subscribe to information from a variety of sources. Client applications would describe their information needs in terms of compositions of basic view functions, and also specify their information quality of service requirements. The dissemination service would use the algebraic properties of the view functions and the quality of service requirements to identify opportunities to merge multiple clients' requests, thereby reducing overall network bandwidth requirements.
- A distributed process control system, where controllers on a network gather measurements at regular intervals from hundreds or thousands of sensors. Process operators, maintenance engineers, and process engineers need different views of the same data, either the raw sensor data or data derived by diagnostic or state estimation algorithms.
- A shared collaborative workspace. Collaborators may have different information needs based on their roles in the collaboration and therefore required different (but synchronized) views of the shared workspace.

Over time, a general-purpose *distributed situation awareness infrastructure* could be developed that could be retargeted to multiple applications.

A second direction is to develop a significant library of view implementations from which applications can be developed. We believe that such a library should be based on well-understood, broadly applicable mathematical concepts such as sets, bags, relations, functions, tuples, etc. These concepts form the basis for the Z and Larch libraries [SPIV98, GH93]. The implementations should be compatible with widely-used object libraries such as the C++ Standard Template Library [SL94] or the Java Foundation Classes [FLAN97].

A third direction is to extend the Active Views mechanisms in various ways. For instance, bi-directional or multi-directional object synchronization should be provided. Currently, object synchronization is uni-directional: the target objects may not be updated except by the synchronizer as a result of updates to the source object. The difficulty lies in the fact that, if the objects are updated independently, their states may diverge, with no general way of reconciling the differences. In general, application-specific reconciliation algorithms are required [GRAY96], but can perhaps be supplied a parameters to general-purpose services as the compression and distance functions have been for the *IQoS* view.

A fourth direction is to extend current database query processing techniques with Active Views concepts. The power of a DBMS would be considerably greater if a query could involve data computed using application-specific view functions. For example, a process plant database could allow a client to query process or equipment health parameters, e.g. "what pumps have had greater than average load over the past month?" Answering such queries requires both access to historical plant data and execution of process- or equipment-specific computations. Examples arise in other application domains such as military intelligence, where answering queries like "what tanks are within 10 miles of here?" requires a combination of image retrieval, image processing, and data fusion. A client posing such a query should perceive no difference, except possibly in response time, between querying stored and computed data. These extensions would be based on the ability to express queries as expressions involving relational algebra operators and application-specific view functions, and to transform such expressions using query optimization techniques to achieve better performance.

Section 3
Block-Based Programming

In this section, we present the P*resto* block-based application programming model and its extensions to support Active Views. The model is based on a data flow programming approach to facilitate construction of continuous multimedia applications. With the model, application functions are implemented as blocks and applications are "programmed" by interconnecting their functional blocks. Thus, the model enables the plug-and-play programming paradigm, making application programming easy and efficient and supporting reuse of application software. Section 3.1 describes the original Presto programming model, and Section 3.2 covers the Active Views extensions.

In the course of extending the extending the Presto programming model to support Active Views, we replaced the existing, custom-built distributed execution environment with one based on COTS products, most notably Iona's Orbix CORBA implementation and Object Design's ObjectStore database management system. Section 3.3 summarizes our experience in doing this.

3.1 Presto Programming Model

A *program* is an application programming model for describing continuous multimedia applications based on the data flow paradigm. It consists of a set of blocks interconnected through data ports by media flow paths. Figure 17 shows an example of a video-capture-process-display program comprising video camera, motion detection, color filter, and display blocks.

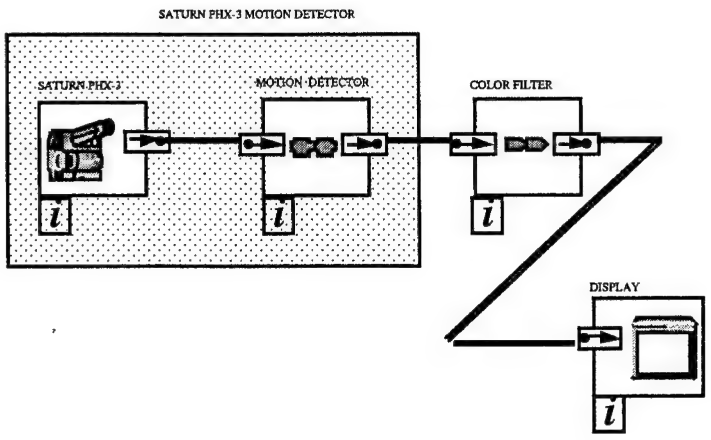


Figure 17. Example Block-Based Program

A *block* consists of:

- A vector of input ports that accept incoming data streams
- A vector of output ports that produce outgoing data streams
- A vector of parameter ports that are used to set operating parameters

- A function that produces output streams, consumes input streams, or transforms input streams into output streams,
- A pair of matrices for data rate and QoS translation between input and output ports.

A block is called *basic block* if its function is coded in a language such as C++ or *composite block* if it is formed by assembling and connecting a set of basic blocks. Figure 18 shows example basic program blocks.

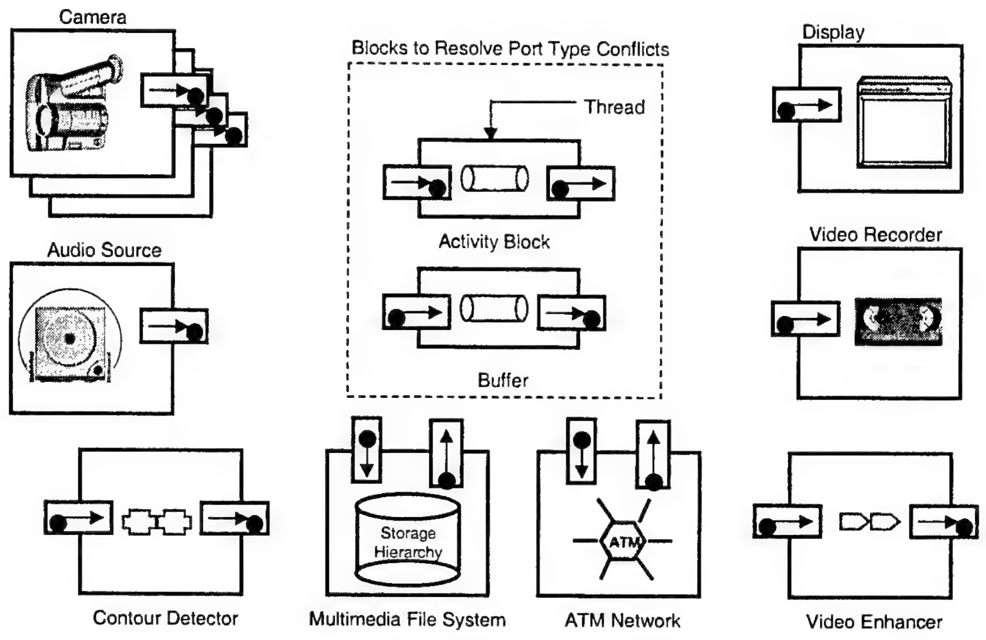


Figure 18. Example Program Blocks

A *port* is an interface of a block that captures interactions with other blocks. As illustrated in Figure 19, it is defined by a data type (JPEG, audio, etc.), a data flow direction (input or output), and a control flow type (push or pull). In push-type control flow, the output port takes the initiative to deliver data to the input port. In pull-type control flow, the input port requests data from the output port. Therefore push output ports and pull input ports are active, whereas pull output ports and push input ports are passive.

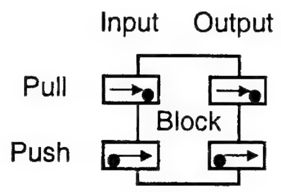


Figure 19. Port Types

Presto includes a Program Development Tool (PDT). Using the PDT, a user can construct a program from a library of basic blocks, composing them by connecting output ports to input ports without regard to push/pull port type compatibility. Such a program is called a *user program*. The Sonata PDT is an evolution of the *Presto* PDT.

P*resto* transforms a user program to a *system program*, as illustrated in Figure 20, by locating push/pull incompatibilities and correcting them by inserting special blocks. If a push output port is connected to a pull input port (both ports are active), P*resto* inserts a buffer block between them. If a pull output port is connected to a push input port (both ports are passive), P*resto* inserts an activity block between them. Separation of the user program from its corresponding system program makes the control flow—the push/pull semantics—transparent to the application programmer and simplifies block-based programming.

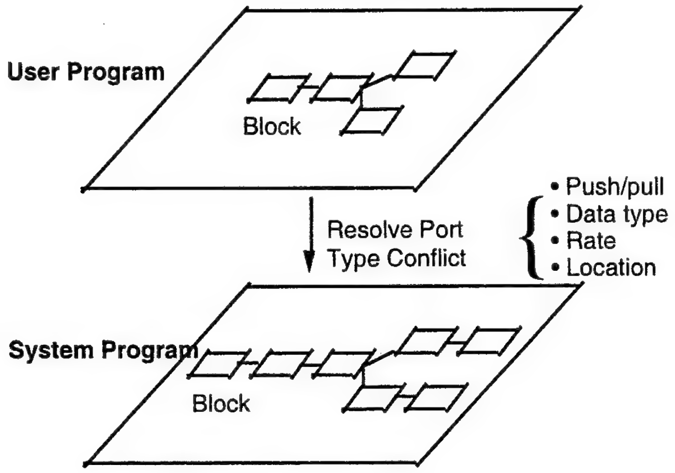


Figure 20. User Program to System Program Translation

The P*resto* block-oriented programming model was inspired by other work on process control applications [SVK93], graphical application development [HAEBE88, INGAL88, KASS92], and commercial simulation tools. However, the previous work did not explicitly address the system integration and timing issues encountered in constructing continuous multimedia applications. P*resto* was unique in providing a software methodology that integrates user-level application development and system-level application execution and in extending the block-oriented programming model with rate and QoS properties to support the temporal constraints of multimedia applications.

3.2 Extensions to Support Active Views

We developed extensions to P*resto*'s data-flow oriented, block-based programming model to support operation flows in addition to data flow, and to handle aperiodic flows in addition to periodic flows. These changes were necessary to implement Active View services, and moved the range of applications well beyond the continuous media applications that P*resto* supported.

In Sonata, we implemented two of the three Active Views mechanisms: view and object synchronization. (We did not implement histories as described in .) Figure 21 shows how a block is structured to realize both view and object synchronization functionality. Each input port consists of an object O_i that receives a flow of requests from the output port of a predecessor block. The object is an interface to a procedure $Derivative_i$ that performs two functions: it updates an internal object O that maintains a materialized state of the view, and it send requests representing any view state changes to all successor blocks. In performing these functions, the derivative procedure can access the materialized states of the predecessor blocks as shown in the figure.

In short, each time a block receives a request from a predecessor block at one of its input ports, it computes the implied change of state of the view and sends it via the output port to successor blocks. This approach propagates state changes through blocks quickly, but can be less efficient than an approach that uses histories to "buffer" state changes arriving at its input ports, and computes the change in the state of the view based on all buffered state changes. Again, we did not implement histories or any concept of information quality of service.

A successor block can be linked to the output port at any time, even while the block is executing. When this happens, the output port sends a sequence of requests to the new subscribing input port that effectively transmits the current state of the materialized view object O. This sequence of requests is constructed by a procedure (not shown) that has access to the internal representation of O. Once the new successor block is "brought up to speed" with the state of the view, any changes of state are sent to it and the other successor blocks as described above.

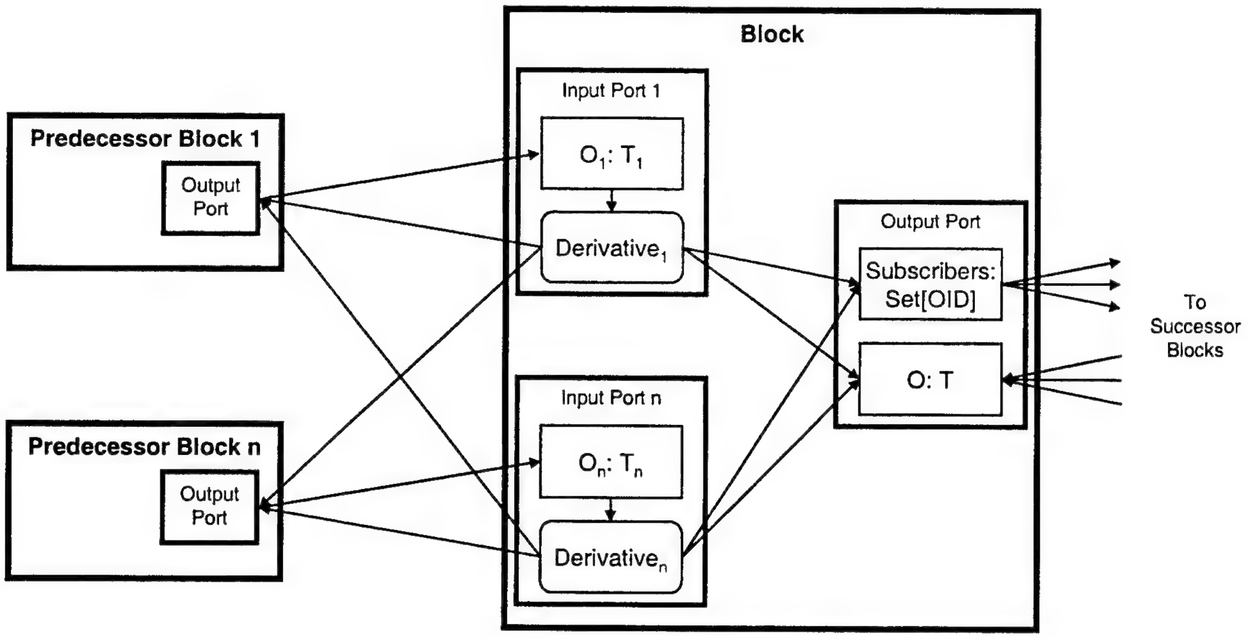


Figure 21. Internal Structure of a Block

3.3 Re-Implementation of the Distributed Execution Environment

For many reasons, including development cost and standards in the application domain, we decided to use commercial off-the-shelf products for distributed object services and persistent object management. Choosing the best products for our particular needs was quite important.

The COTS products that we picked were Iona's Orbix and Object Design's ObjectStore. The application domain standards mandated the use of CORBA [OMG95]. We chose Orbix in view of its dominant presence in the Unix environment. The selection of an object database was much more difficult–since there are a plethora of products, but no standard. We chose ObjectStore, based on a survey of OODBMSs [PS97].

The choice of these products strongly affected the design and implementation of our system.

Object systems interoperability—We were faced with the problem of handling three different object systems, i.e., the one that comes with the programming language (C++), the distributed object management system (CORBA/Orbix), and the object database system (ObjectStore). There is enough difference in the three approaches and their abilities that we had to consider interoperability issues. For example, CORBA and ObjectStore have different object granularities–we could model an ObjectStore collection as a CORBA object, but the individual objects that comprise such a collection couldn't be easily fit into the CORBA model.

The CORBA model defines an object by its interface, while the ObjectStore model is tied to the implementation of an object. This tension both helped and hampered us. It helped us in that the two systems affected different parts of the design, and changes made for one did not affect the other too much. It hampered us by increasing the number of variables to deal with in the overall design and implementation.

Distribution—A related issue was that CORBA and ObjectStore have different models of distribution. This strongly affected our model of distribution.

CORBA was mainly useful in making our framework support distributed applications. We discovered that, since our framework imposed certain requirements on the style of code written, making it distributed was relatively easy. The transition to using CORBA was reasonably painless, once we understood the conceptual differences in the object models.

Database design—ObjectStore's implementation exposed *reference* semantics, which are not naturally modeled by conventional database views. We thus had to make some basic changes to our model to accommodate this.

Choice of an object-oriented language—The choices were C++ and Java. C++ has more mature off-the-shelf products, but Java claims to be the language of choice for distributed objects.

We felt that C++ should be the language used, since most of the COTS products for Java weren't mature enough at the time.

We used C++ templates extensively to enforce interfaces, without being too dependent on a class hierarchy to provide it. This was of great help when we had to modify our class hierarchy to accommodate Orbix and ObjectStore. Also, the concept of template *traits* helped us hide the differences in accessing persistent and non-persistent objects, and differences in accessing local and remote objects.

Real-time and QoS—We envisioned the application being used interactively, with the presentation of multiple related data-types at the same time. For example, one window would show a map, another would show a video of the same location, and another would list the resources in that area.

In enforcing real-time and QoS requirements [WS96], we were strongly limited, since we had no good model of the behavior of the COTS products that we used. Due to this, we have not yet addressed these issues in our implementation.

3.4 Conclusion

From conception to design to implementation, we learned and re-learned many of the lessons in [SF97]. We found that just deciding to use COTS technologies wouldn't immediately solve all our problems in the areas they address. This framework benefited strongly from being incrementally designed from P*resto.*

In conclusion, we believe that the use of COTS technology has added value to our project, and the benefits of using them outweigh the costs. The support for persistence and the support for distribution have allowed us to enhance and enrich our framework.

Section 4
Application Development Tools

4.1 Introduction

This section is a report of the Program Development Tool (PDT). It describes the overall project environment for which the toolkit is built, specific requirements of the tool, design and implementation issues. It also proposes avenues for further research.

The remainder of this section gives an overview of the multimedia project and focuses on the aspects specific to the Program Development Tool. Section 4.2 discusses the need for such a toolkit. Section 4.3 describes the design issues that have been tackled by the group. The implementation issues are discussed in Section 4.4. There are many areas in which future research is possible. These are described in Section 4.5 for those who would like to continue work on the toolkit. Finally, Section 4.6 concludes the report by highlighting the key aspects of the project and the lessons learned.

4.1.1 PDT Overview

There has been an increasing interest in the area of multimedia systems, causing it to emerge as an independent discipline of study in computer science and engineering. Consequently, a number of commercial products and prototype systems with various levels of sophistication and abstraction have been built in the last few years. However, it has been observed that the current programming paradigm for developing multimedia software needs some improvement [PATEL95], and [TG95].

Sonata's programming model is to visualize a multimedia application as a directed graph, wherein the nodes represent common generic multimedia blocks (Camera, Display, etc.) and the edges depict the flow of multimedia streams. The nodes in the graph, called *blocks*, represent operations that modify streams as they flow through them. The operation parameters of a block are specified through parameter ports. The multimedia streams flow through data ports.

The development of a programming paradigm is typically accompanied by the design of new languages and an associated set of development tools. Based on experiences in software engineering and language design, it is hoped that these tools will make it easier to write Sonata programs (or applications), and consequently help in enhancing Sonata's programming language and interface.

The Program Development Tool is part of the Sonata development toolkit. It facilitates the creation and modification of Sonata applications using the block-based programming paradigm. Using the Program Development Tool is the first step in the implementation of a multimedia application. The output of the tool is transferred to the User Interface Design Tool (another part of the development toolkit) before being executed by the Sonata run-time environment.

4.1.2 Integrated Toolkit Environment

In order to exploit the ease and power of Sonata, a high-level application toolkit has been designed. The three basic needs from the toolkit have been defined as

- Specification of blocks, ports and connection structures
- Specification of user interface component parameters
- Verification and analysis of applications

In response to the above requirements, the application development toolkit consists of three separate yet integrated tools, namely

- Program Development Tool (PDT)
- User Interface Development Tool (UIDT)
- Program Analysis Tool (PAT)

The PDT facilitates the creation and maintenance of Sonata applications. It allows the user to specify the blocks and the interconnection between them in an application. The UIDT is used to interactively define the behavior of the user interface blocks. The PAT is intended to serve as an error detection and correction tool for the application.

The advantages of having an integrated toolkit are numerous. The PDT frees the programmer from the burden of specifying a graphic structure in a non-graphical language. The UIDT eliminates the need for writing customized user interface code. The PAT is targeted to save valuable time by automating the error detection mechanism.

One of the key issues to be discussed later is that of implementation. There are a few requirements that have been imposed on the toolkit. The main issues here are

- Interoperability between tools
- Ease of use (minimal learning curve)
- Portability

Section 4.4 describes these issues in greater detail and provides the key decisions that have been made regarding the implementation.

4.1.3 Programming Model

Continuous media applications can be modeled using the data flow paradigm [TAC+95], [SVK93]. In such a paradigm, an application program consists of a collection of data *pipes* that regulate the flow of continuous media streams through functional *blocks*, which encapsulate functions or operations that are performed on continuous media streams. The pipes and blocks are depicted by a directed graph to achieve the overall functionality of the application.

Typically, the data is generated by *source blocks* (camera, file system, microphone, etc.) and presented to the user by *sink blocks* (display, speaker, etc.). Between the source and sink blocks, pipes connect the intermediate blocks various processing (image recognition, thresholding, synchronization, etc.) functions. Figure 22 illustrates an example application.

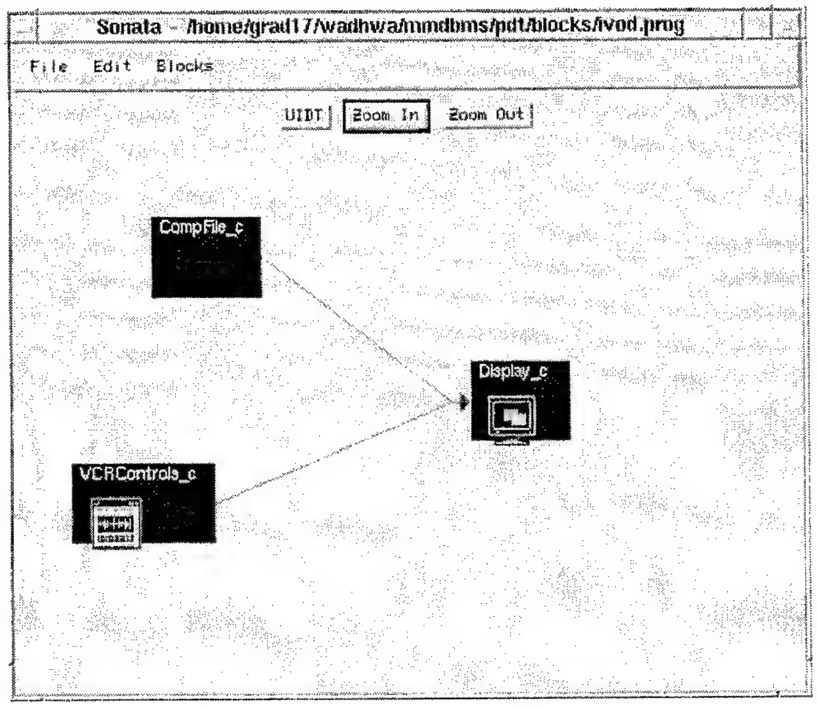


Figure 22. Interactive Video-on-Demand

The application shown above is an interactive video-on-demand system. The user selects the multimedia file to view, and then manipulates it using the VCR Control block. The CompFile_c block supplies the continuous media stream of information while the Display_c block gets the data and displays it to the user. The VCRControls_c block allows the user to browse through the CM frames.

The Sonata programming and run-time environment supports the construction and execution of distributed multimedia applications. It allows the construction of new applications from a set of primitive blocks via a basic programming language.

A primitive block consists of the following components

- An associated piece of code that implements its functionality
- A set of *data ports* that are used to pipe in/out multimedia streams
- A set of *parameter ports* that are used to set its operating parameters

The source code that implements the functionality of the block is totally generic. It does not have any application specific routines. For example, the CompFile_c block in the application shown in Figure 22 is not aware of the fact that its output is being sent to the Display_c block. The application specific parameters of a block are set by its parameter ports.

Data ports, used to input/output data streams from blocks, consist of two types according to their functionality: *input ports* and *output ports*. Furthermore, when two blocks are connected to each other, the data ports can also be classified as *push* or *pull* depending on which block is responsible for the data exchange. If a receiving data port is responsible for the action of actively grabbing data, then it is said to be a pull port. Conversely, if a connected output port is responsible for actively sending the data, then it is said to be a push port.

Parameter ports are used to set the operating parameters of a block. There are various classes of parameters that a particular block may have. A display block, for example, will have ports that convey its geometric information, such as its size, location and screen offset. Ports can also convey Quality of Service parameters [WS96]. Parameter ports may be either static (value specified at creation time) or dynamic (value specified at run time).

4.2 The Need for PDT

With the program development tool, the user works at a convenient level of abstraction—the block. At this level, the details of implementation are omitted. Hence, the user is allowed to focus on the more creative aspects of multimedia programming. The implementation details will be specified later in the program creation process.

PDT provides an easy-to-use interface with a minimal learning curve. The operations that can be performed are intuitive and very easy to understand. At the same time, the tool is flexible and versatile. It can handle the most complicated application program as easily as it can handle a simple one. It is also designed to be as comprehensive as possible, thereby allowing the user to complete the entire application without leaving the toolkit environment.

Prior to the implementation of the program development tool, a list of requirements was laid out. The specifications essentially stated what was expected of the toolkit. The remainder of this section describes the requirements in greater detail.

4.2.1 Requirement Specification

The tool is required to achieve a list of suggested features, as indicated below.

2.1.1 Block/Port Requirements

The tool should allow the user to:

- Create a new block,
- Establish a connection between two blocks,
- Delete an existing block,
- Delete a connection between two blocks,
- Modify the data ports of an existing block,
- Modify the data ports of an existing connection,
- Add a parameter port to a block,
- Delete a parameter port from a block,
- Modify the parameter ports of an existing block,
- View the ports and connections of a block.

2.1.2 Editing Capabilities

All blocks and connections are drawn on an application canvas. The tool should allow the user to:

- Select a set of blocks on the canvas,
- Deselect a set of blocks on the canvas,
- Copy the selected blocks to the clipboard (an offscreen buffer),
- Cut the selected blocks to the clipboard,
- Paste the contents of the clipboard,
- Move a set of blocks around on the canvas,
- Refresh the canvas,
- Undo the last change.

4.2.2 Input-Output Specification

All application programs are stored in a file. Hence, the tool should be able to recognize and work with the specified file format. Furthermore, it should allow the user to

- Save the current session to a file,
- Open an existing application from a file,
- Start up a new application program file,
- Quit with/without saving changes.

The format of an application program is divided into five different parts. The first part consists of a list of machines on which the application will execute. For each machine, the canvas size and background color is stored. The second part is a list of blocks and its definitions. The third part shows the connections between the blocks and specifies the ports at which the connection is being made.

The fourth and fifth part contains a list of parameters for each of the blocks. For each parameter, its type and value is stored.

4.2.3 Integration with UIDT

The Program Development Tool is closely knit with the User Interface Development Tool. The UIDT allows the specification and modification of the properties of user interface blocks, such as camera, display, speaker and microphone. The parameters of these blocks can be modified interactively using the UIDT.

Because of the close connection between the two tools, one of the requirements stated that the two tools be integrated. The user should be able to interoperate between the two tools. Both the tools share a common file format. The implementation of the integration will be discussed more specifically in Section 4.4.

4.3 Design Issues

As with any good project schedule, a significant amount of time was spent in the design of the Program Development Tool. The key issues discussed were compatibility with UIDT and block-based design. The design phase of PDT included the following tasks

- Determination of program structure
- Identification of block hierarchy
- Identification of objects and methods

In determining the program structure, it was decided to modularize the program as much as possible. The input and output functions are kept separately from the main program, so that future modification is easy. A generic library of functions is made available, so as to enhance re-use of code. Finally, the design is very flexible and allows for easy maintenance and modification.

For the most part, the identification of objects and methods was trivial. The complications arose in case of composite blocks. Blocks may be composed of other blocks.

The entire design document is attached at the end of this report as an appendix. The remainder of this section discusses the key points that were encountered in the design process.

4.3.1 Block and Port Design

A set of connected blocks can be part of a composite block. The composite block may then be used as part of another application. This hierarchical organization of blocks leads us to using a tree structure in its representation. To give an example, the CompFile_c block shown in Figure 22 is actually a composite block and its internal representation is shown in Figure 23.

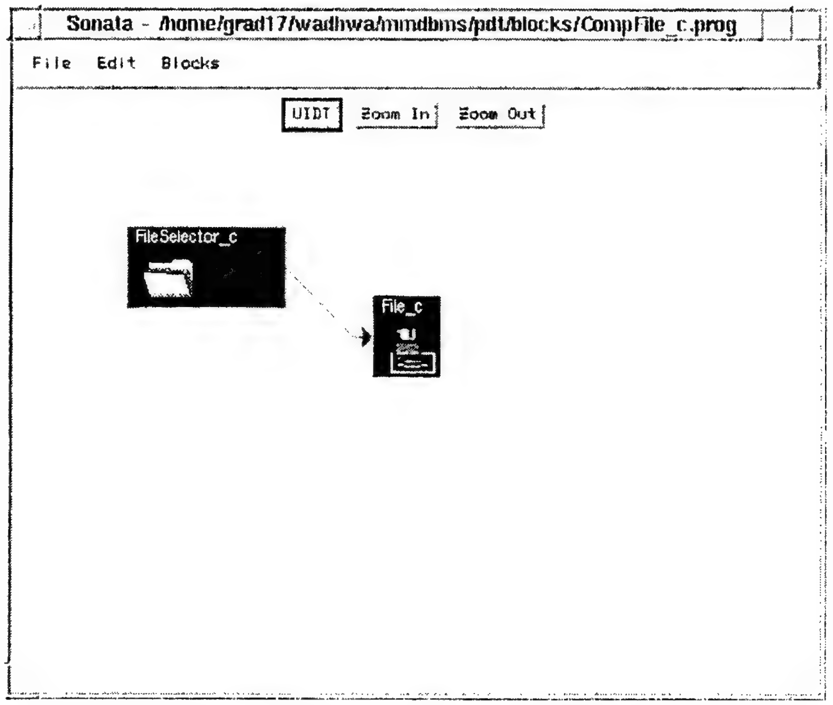

Sonata - /home/grad17/wadhwa/mmdbms/pdt/blocks/CompFile_c.prog
File
Edit
Blocks
UIDT
Zoom In
Zoom Out
FileSelector_c
File_c

Figure 23. A Composite Block

With the representation of composite blocks, there is the notion of zooming in and out. Zooming into a block exposes its internal representation. Zooming out of an internal representation shows the block level view. The program in Figure 1 is the zoomed out version, which is the default view of an application. The zoomed in version, with the exposed internal representation of the composite block is shown in Figure 24.

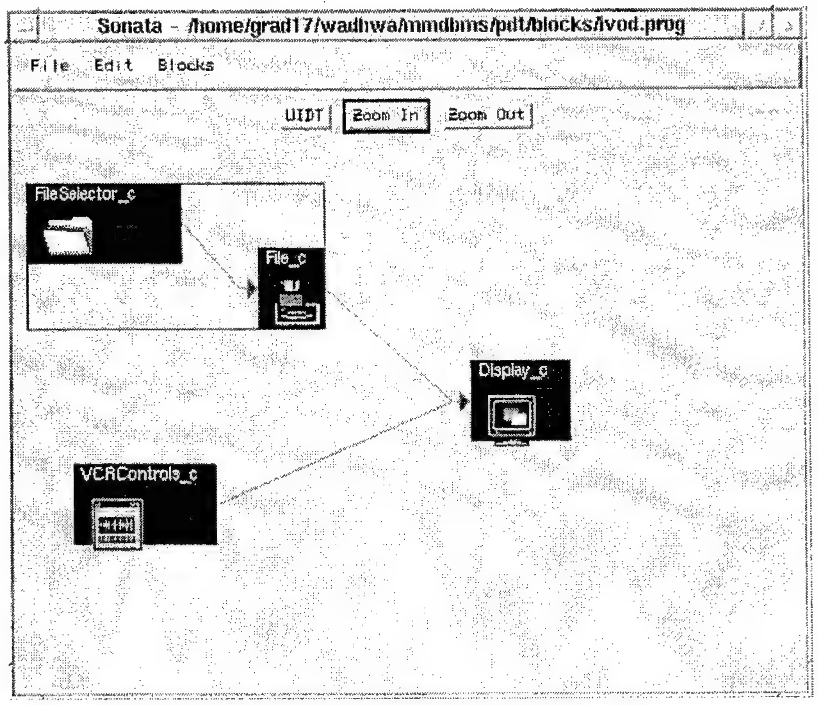


Figure 24. Zoom In View

As part of the design process explained earlier, an identification of the objects for blocks and ports was done. A class Block was defined that contained the following information:

- Block number: Integer
- Block name: String
- Associated block source code: Text
- List of Parameter Ports: List
- List of Data Ports: List

A class called Port was also defined that had the following information:

- Port name: String
- Port type: String
- Port value: String

Each instance of the class Block contains a list of Port objects. The object definition contains the relevant methods that are required to add, delete or modify a particular port.

4.3.2 Connection Design

There are two different ways of storing the connection information. The first is to flag the ports in the corresponding blocks and define the flags as a connection. In this case, the block object defined above would contain an additional list of flags for the data ports.

The second way is to have a separate object called a Connection object in which input and output block tags are kept. The Connection structure is maintained separately from the Block structure. It has its own methods for addition, deletion and modification.

The former approach has a couple of drawbacks. Information about the connection is stored twice (at the input and output block object). When a connection is to be deleted, it is ambiguous as to which block should take the responsibility. All modifications would have to be performed at two places instead of one. The only advantage gained by having the connection information in the same object is that it avoids the need for having a separate object.

With all the disadvantages of the former method, the latter approach was finally selected. The connection object is a separate identity and it contains its very own information, which is

- From Block Number: Integer
- From Block Port Name: String
- To Block Number: Integer
- To Block Port Name: String

The advantages of the latter approach are numerous. The maintenance of connection information is very easy. Deletion of a particular connection involves removing only one instance. There is a single object repository of all information pertaining to the connections in the application.

The application program file is now made very simple. It is an object that contains quite simply a list of blocks and a list of connections. The block object contains all the details of its data and parameter ports. The connection object contains all the details of the connection structure in the application.

4.4 Implementation Issues

Having done the design, the implementation phase should really have been trivial. However, there are a lot of issues that came up during the project. Most of these issues were to do with the user interface. Most of the feedback was incorporated into the program and then resubmitted for appraisal. This feedback loop continued for most of the project, especially after the first draft of the toolkit was released.

Apart from the user interface issues, there were other relevant concerns like the language of implementation and compatibility between the other tools. The remainder of this section discusses these issues in greater detail.

4.4.1 Language Issues

The language of implementation was chosen with several different factors in mind. It was decided to use the language that best fit the requirements. The requirements of the language/system were

- Portability
- Ease of maintenance of code

- Object-oriented methodology
- GUI capabilities

The biggest factor among all is portability. The toolkit is designed to work with any platform. Communication between the tools is done using a file, so the language must provide the ability to work with files. To provide for future modifications, the language must be easy to understand and read. This helps in the maintenance of the source code. Most object-oriented languages have this advantage.

Another reason to use an object-oriented language is because the design of blocks and ports uses objects. A block can be defined as a class and each instance would be an object. Ports can be similarly defined as classes. Connections can be implemented using classes as well, where the block tag would essentially correspond to the object reference.

It would be highly beneficial if the chosen language also contained GUI capabilities. Most user interface APIs are known to be platform specific (e.g. Motif with X, Visual C++ with PCs). If the language itself had built-in GUI routines, then the language is typically platform independent (e.g. Tcl/Tk).

The language finally chosen was Java [GYT96]. It has all of the features described above and more. Java seems to be ideally suited for the task. The other languages considered were Tcl/Tk [OUSTE93] and C++. The drawback of Tcl/Tk is that it's a scripting language with insufficient object-oriented features while C++ is not too portable and does not have many platform independent GUI capabilities.

With the implementation in Java, the toolkit is portable to any system that supports the Java Virtual Machine.

4.4.2 User Interface Issues

There was a lot of time spent on designing the user interface of the Program Development Tool. After the implementation, the user interface was shown to the potential users and their feedback was considered for the next revision. This process was implemented a couple of times until the users were satisfied.

The final layout of the user interface was found to be pretty appealing. It is easy to use, flexible enough to support enhancements and powerful enough for complicated applications.

4.4.3 Integrated Toolkit Issues

The program development tool is part of an integrated toolkit. There is a need to ensure seamless interaction with the other tools. The file format of the program depends on the mode of interaction with the other two tools.

The file format that was used is given below. The format captures all the information required for the program application. It can be easily used to represent the application with all the three tools. Note that there is no toolkit specific information represented here.

```
# Canvas Information
<Canvas #> <Canvas Name> <Canvas Location> <Canvas XOffset>
<Canvas YOffset> <Canvas Width> <Canvas Height> <Canvas
LeftFooter> <Canvas RightFooter> <Canvas Color Red> <Canvas
Color Green> <Canvas Color Blue>
-1 NoFrame NoLocation -1 -1 -1 -1 NoLeftFooter NoRightFooter
-1 -1 -1

# Block Information
<Block #> <Block Name>
-1 NoBlock

# Connection Information
<From Block #> <From Port Name> <To Block #> <To Port Name>
-1 NoPort -1 NoPort

# Parameter port information
<Parameter Name> <Block #> <Parameter Type>
NoParameterPort -1 -1

# Parameter port information
<Parameter Name> <Parameter Value>
NoParameter NoValue

# End of File Marker
-1 NoFile
```

For example, the file format of the application in Figure 22 is given below.

```
0 Canvas0 rawana 50 50 400 400 presto presto 255 255 255
-1 NoFrame NoLocation -1 -1 -1 -1 NoLeftFooter NoRightFooter
-1 -1 -1
3 Display_c
4 VCRControls_c
5 CompFile_c
-1 NoBlock
4 VCROutput 3 VCRInput
5 Output 3 Input
-1 NoPort -1 NoPort
DispLoc 3 Location
DispX 3 XLocation
DispY 3 YLocation
DispW 3 XWidth
DispH 3 YHeight
VCRLoc 4 Location
VCRX 4 XLocation
VCRY 4 YLocation
VCRW 4 XWidth
VCRH 4 YHeight
NoParameterPort -1 -1
DispLoc rawana
DispX 200
DispY 200
DispW 100
DispH 100
VCRLoc rawana
VCRX 200
VCRY 100
VCRW 100
VCRH 10
NoParameter NoValue
-1 NoFile
```

4.5 Proposal for Further Research

Although the integrated toolkit is stable, there is a lot of potential work that can be done in this area. This section describes the avenues in which future research is possible.

4.5.1 Interaction with CMT

The Continuous Media Toolkit (CMT) is a system developed to support continuous media flow. One of the ongoing and future directions of this project is to make the Program Development Took interact with CMT.

CMT provides an extension to the Tcl shell. The program file can be translated to a Tcl program by means of a translator. The job of the translator would be to interpret the various blocks in the application and try to use the corresponding Tcl commands.

Once the translator has successfully translated the file, the resultant Tcl program can be executed under the Tcl shell. There are a couple of hurdles in this task though. The biggest one is to achieve distribution. There is a slight incompatibility in the way CMT handles distributed applications versus the way PDT handles it. This issue needs to be resolved before an efficient translator can be built.

4.5.2 Implementation of PAT

The Program Analysis Tool is intended to serve as a potential performance analysis and compatibility verification tool. There are many advantages of having the PAT. Earlier error detection leads to a large gain in time. The automation of the error detection mechanism minimizes human error.

The complexity of the PAT can be of varying degrees. The most basic analysis would be to check the compatibility of connections between ports. On the other hand, more complex analysis would involve type checking and error correction.

The implementation of the PAT is a large area of further research. It can be very advantageous to the overall system.

4.6 Conclusion

The program development tool is part of an integrated toolkit that allows the user to build distributed multimedia applications. The tool is fairly easy to use, powerful and flexible. There is a need for the program development tool to allow the user to concentrate on the systems aspect of the application rather than having to worry about the underlying blocks.

The PDT provides the basic editing capabilities on the blocks and ports. Apart from that, it allows the user to work on a wide variety of functions that are comprehensive. The user interface is powerful. It can handle the most complicated programs as easily as it can handle the basic ones.

There is a large scope of further research in this area, especially with regards to interaction with the Continuous Media Toolkit (CMT) and the implementation of the Program Analysis Tool (PAT).

It is hoped that the experiences in this project will benefit future systems in this area. The project has been a tremendous help in our understanding of the development of multimedia systems.

Section 5
Continuous Media Server

5.1 Introduction

Continuous media (CM) server has recently been a hot research topic for several reasons. First of all, network speed is increasing, thus, in the near future, services like Video on Demand, Teleconferencing, Distance Learning are very likely to be popular in everyday life. But, given the limitations of current network bandwidth, straightforward TCP implementations are not suitable for such bandwidth-sensitive applications.

TCP has its own flow control mechanisms, error detection and retransmissions, all of which add extra time as well as network bandwidth overhead to the transmission. This causes unexpected and unpredictable delay and jitter time when transferring CM data, while timing is one of the most critical requirements of CM applications. Most CM applications don't need highly reliable transmission. Losing some frames is less important than having too much delay jitter or losing synchrony between streams. A convincing fact is that a typical TCP connection bandwidth is 2.6 Mbps on a LAN, 580 Kbps on the MAN, and 104 Kbps on the WAN. While for UDP, they are 9 Mbps, 5 Mbps, and 1.2 Mbps respectively on LAN, MAN and WAN. Obviously, TCP is not a good candidate for high bandwidth media streaming.

Given that observation, the natural questions are:

- Is UDP suitable for CM applications?
- How good/bad is it?
- What are the criteria (QoS) to for evaluating it?
- If it is bad, how do we reduce the lossy property of UDP while still making use of its higher capability of bandwidth for applications that are speed-sensitive like CM servers?

Secondly, the loss of UDP packets sending over a network is usually caused by buffer overflow. We have tested out that if a sender keeps pushing UDP packets onto the network, even if network bandwidth is good enough to handle it, there are still a lot of packets lost because the consuming time of the receiver is quite large. This applies perfectly to client/server kind of application. For example, with video streaming, the consuming time of a client depends largely on the capability of video cards, which are not always good. This is even worse for audio streams, an 8 kHz audio stream (e.g. telephone voice) can be played only at 64 Kbps. This delay could cause lots of dropped UDP packets if there is nothing done at the client and/or the server. Moreover, buffer overflow can occur at any of the network switches as well. How do we detect and deal with the fact that the client is good enough to handle data but network congestion limits the stream reliability.

Next, the fact that human ears are a lot more sensitive to interrupts in voice than human eyes to interrupts in video frames, raises up another natural question: how do we deal with loss of UDP packets in loss-sensitive stream like audio?

Moreover, given limited resources like network bandwidth, I/O time (disk seek, latency time), memory capacity, and CPU time, the capability of a CM server will obviously be reduced as the number of streams (clients) increased. A best effort strategy is simple, but a preferred policy is to deny a request if the CM server knows that it is not able to handle the request. An appropriate admission control algorithm must be adopted for this purpose. Even if all the above problems have been solved, inter-stream and intra-stream synchronization are questions next to be answered.

Lastly, we ported our socket-based CM server with CORBA. We used version 2.0 of Orbix from IONA Technologies as the CORBA implementation. These versions replace all C socket calls with stubs and skeletons generated from a pair of CORBA interface definition language (IDL) specifications. The IDL specification uses it sequences parameters for the data buffer rather than string parameters, which are a bit slow. Due to the higher fixed overhead of CORBA such as demultiplexing and memory management, this version shows much lower performance.

In this section, we present a design and implementation of a QoS-driven CM server based on part of the analysis presented above.

5.2 Related Works

This section summarizes some work that have been done related to CM delivery mechanisms, CM server architectures, network protocols for multimedia data, and some admission control algorithms known in the literature. Many research systems use TCP/IP for media transmission. As we have discussed in the previous section, UDP seems to be a better choice for bandwidth sensitive applications. One of the most popular CM delivery tools is the Continuous Media Toolkit (or CMT for short) [SMITH94, PATEL95]. CMT has a Tcl/Tk interface that allows quick prototyping, and makes multimedia programming easier. In CMT, media data within a stream is read from files by MediaSource object associated with that particular stream. Then, the data is passed to PacketSource, which then break the frames into packets and send them to PacketDest objects. PacketDest resembles the frames from the packets received and invokes the CMPlayer after some certain interval for displaying (to screen or to audio device). CMT currently supports SUN au, MJPEG, and MPEG. However, CMT has its own network protocol and dropping strategies, which are sometime not suitable for certain kinds of requirements (e.g. synchronization). Also, it makes it difficult to test out other protocols, dropping policies, and such.

Z. Chen et al. described Vosaic [CTC], an extension of Mosaic to support Video and Audio on the web. Vosaic supports playing back on the web browser. The approach taken in Vosaic is similar to that of CMT. It uses Video Datagram Protocol (VDP), which is a retransmission based protocol like Cyclic-UDP, but simpler. They have shown that VDP performs reasonably well and that multimedia on the current Internet is possible. However, Vosaic has a really simple admission control protocol. Basically, it only restricts the number of concurrent streams connected to the web server. Moreover, Vosaic is multiprocessing based, which is not really a good choice in some cases.

With respect to QoS metrics, Steinmetz has done impressive work on surveying as well as specifying interesting factors that affect human perception on various aspects of multimedia

[STE96]. However, Steinmetz did not consider the lossiness of the streams, hence made the parameters unsuitable for evaluating lossy CM applications. Wijesekera [WS96] defined a different metrics that are more suitable for the lossy nature of UDP based CM communication. Our experimental evaluation is based on his metrics.

5.3 Motivation & Objectives

In order to achieve high performance in designing our CM server, we should consider the resource constraints as well as the properties of CM streams. CM streams have their own features and special QoS metrics. Since we decide to design our CM server and clients based on the lossy UDP, we should adopt the QoS metrics that are suitable for the lossy protocol. These QoS metrics play an important role in our QoS-driven CM server (in particular, QoS Manager).

In this section, we discuss the resource constraints, the QoS metrics for CM streams, and our objectives in designing our CM server.

5.3.1 Resource Constraints

The performance of a CM server is constrained by the resources it can make use of. The CM server is I/O bound. The most important resource is the I/O bandwidth. Usually CM streams are stored as files in disks. A disk is divided into blocks. Disk rate decides the maximum block rate for a file system. The achievable disk I/O bandwidth is constrained by the structure and scheduling algorithm of the file system. When multiple CM streams are stored in the same disk, this achievable disk I/O bandwidth decides the maximum number of concurrent CM streams.

Another resource is the available buffer size for CM streams. The operating system's virtual memory is much large than its physical memory. However, we want to keep our buffer inside the physical memory to save the valuable disk I/O bandwidth. So the system's physical memory of the operating system is an upper bound for the total buffer size. The available buffer size is less than this because the operating system and user program occupy some memory.

Since our CM server communicate with clients by network, the network bandwidth is another constraint for our CM server. There is one connection between the server and a client. Given the fact that nowadays the network bandwidth is much higher than disk bandwidth, we assume that the disk bandwidth gives a more tighten bound than the network bandwidth.

The communications between our CM server and clients are based on UDP/IP protocol instead of TCP/IP. UDP is a lossy protocol, so some Logical Data Unit (LDUs) [STE96] may be lost. Actually, for CM streams, such as video and audio, some losses are tolerable. This requires us to define some kinds of quality of Service parameters to measure the losses.

5.3.2 QoS Metrics

Wijesekera [WS96] defined a set of metrics that are suitable for the lossy nature of UDP based CM communication. Continuity of a CM stream is measured by three components; namely rate, drift and content. For the purposes of describing these metrics, we envision a CM stream as a flow of data units (referred to as logical data units—LDUs in the uniform framework). The ideal rate of a flow and the maximum permissible deviation from it constitute our rate parameters.

Given the ideal rate and the beginning time of a CM stream, there is an ideal time for a given LDU to arrive or to be displayed. Given the envisioned fluid-like nature of CM streams, the appearance time of a given LDU may deviate from this ideal.

The rate variations can be measured more accurately by drift parameters. Our drift parameters specify aggregate and consecutive non-zero drifts from these ideals, over a given number of consecutive LDUs in a stream. For example, the first four LDUs of two example streams with their expected and actual times of appearance are shown in Figure 25. In the first example stream, the drifts are respectively 0.0, 0.8, 0.2 and 0.2 seconds; and accordingly it has an aggregate drift of 1.2 seconds per 4 time slots, and a non-zero consecutive drift of 1.2 seconds. In the second example stream the largest consecutive non-zero drift is 0.2 seconds and the aggregate drift is 0.3 seconds per 4 time slots. The reason for a lower consecutive drift in stream 2 is that the unit drifts in it are more spread out than those in stream 1.

In addition to timing and rate, ideal contents of a CM stream are specified by the ideal contents of each LDU. Due to loss, delivery or resource overload problems, appearance of LDUs may deviate from this ideal, and consequently lead to discontinuity. Our metrics of continuity are designed to measure the average and bursty deviation from the ideal specification. A loss or repetition of a LDU is considered a unit loss in a CM stream.

The aggregate number of such unit losses is the aggregate loss of a CM stream, while the largest consecutive non-zero loss is its consecutive loss. In the example streams of Figure 25, stream 1 has an aggregate loss of 2/4 and a consecutive loss of 2, while stream 2 has an aggregate loss of 2/4 and a consecutive loss of 1. The reason for the lower consecutive loss in stream 2 is that its losses are more spread-out than those of stream 1.

A human's response to video and audio is quite interesting. According to Wijesekera's work, up to 23% of aggregate video loss and 21% of aggregate audio loss are tolerable. The acceptable values for consecutive loss of both video and audio are approximately 2 LDU. Up to about 20% of video and 7% of audio rate variations are tolerable. They give an upper bound for ADF and CDF.

5.3.3 Objectives

It is a reasonable requirement for the CM to guarantee all the QoS parameters defined above. When the load of CM server is low, it is possible to meet this requirement. But when the number of concurrent CM streams increase in the CM server, it becomes difficult to guarantee all the QoS parameters. It is easy to guarantee only ALF and CLF by delaying the following LDUs, which makes the ADF and CDF unacceptable. On the other hand, it is easy to guarantee only ADF and CDF by delaying the early LDUs and dropping the late LDUs, which makes the ALF and CLF unacceptable.

Given certain resources, we want to support as many as possible CM streams whose QoS parameters are all acceptable. There are three approaches: to guarantee ALF and CLF first, to guarantee ADF and CDF first, and to compromise between to guarantee ALF/CLF and to guarantee ADF/CDF.

Furthermore, as more and more clients require CM streams, the quality of service of CM server will degrade. It is a good choice to make it degrade gracefully. To guarantee each client to be served with some reasonable quality, admission control is also necessary.

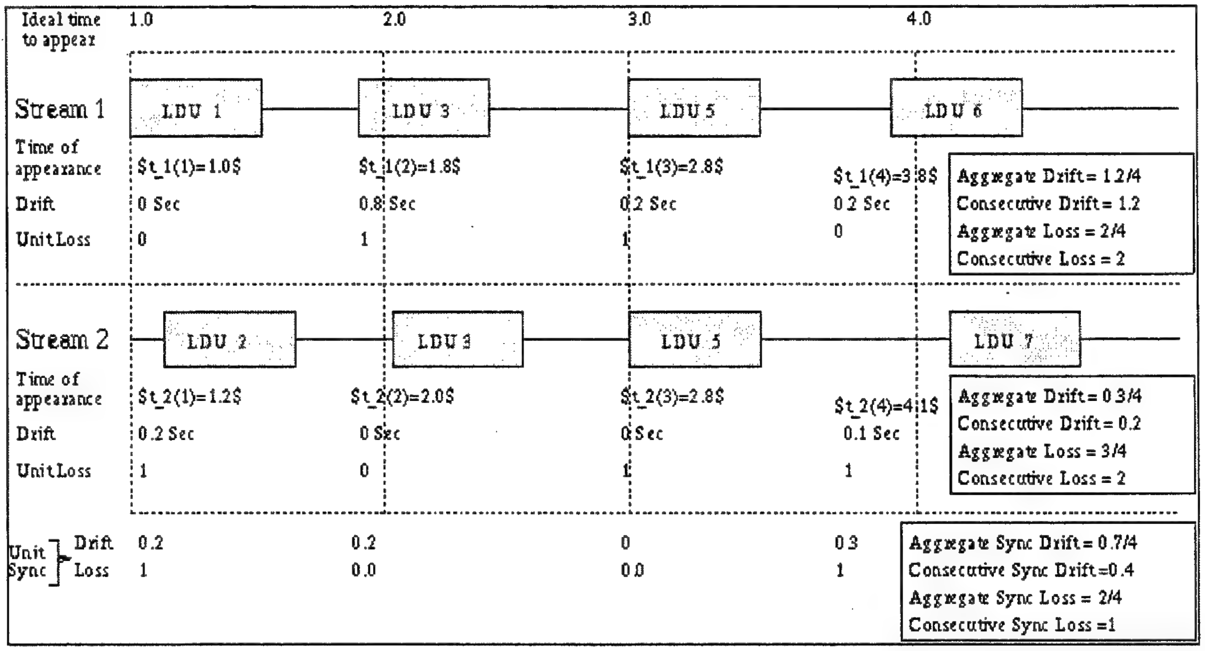


Figure 25. Two Example Streams Used to Explain QoS Metrics

5.4 Design of CM Server

In this section, we describe our CM server system [LSNS98] and its architecture. Its admission control policy and QoS-driven dropping mechanism are also presented in this section.

5.4.1 Overview of Architecture

The CM server system is a typical client/server application. It includes one CM server and some CM clients. The CM server may serve the clients concurrently. Figure 26 depicts the architecture of a CM server that plays out multiple (single) streams to the requesting clients across the network.

The CM server has four kinds of components. The Network Manager responds to clients' connection requests. The QoS Manager is responsible for admission control and I/O scheduling. Each Proxy Server communicates with a client, receiving CM stream operation requests and sending CM data by network. Each I/O Manager reads out CM data from disks for a proxy server. There are as many proxy servers and I/O managers as CM clients.

The CM client is relatively simple compared with the CM server. It has two main components, one Client N/W Controller and one CM Player.

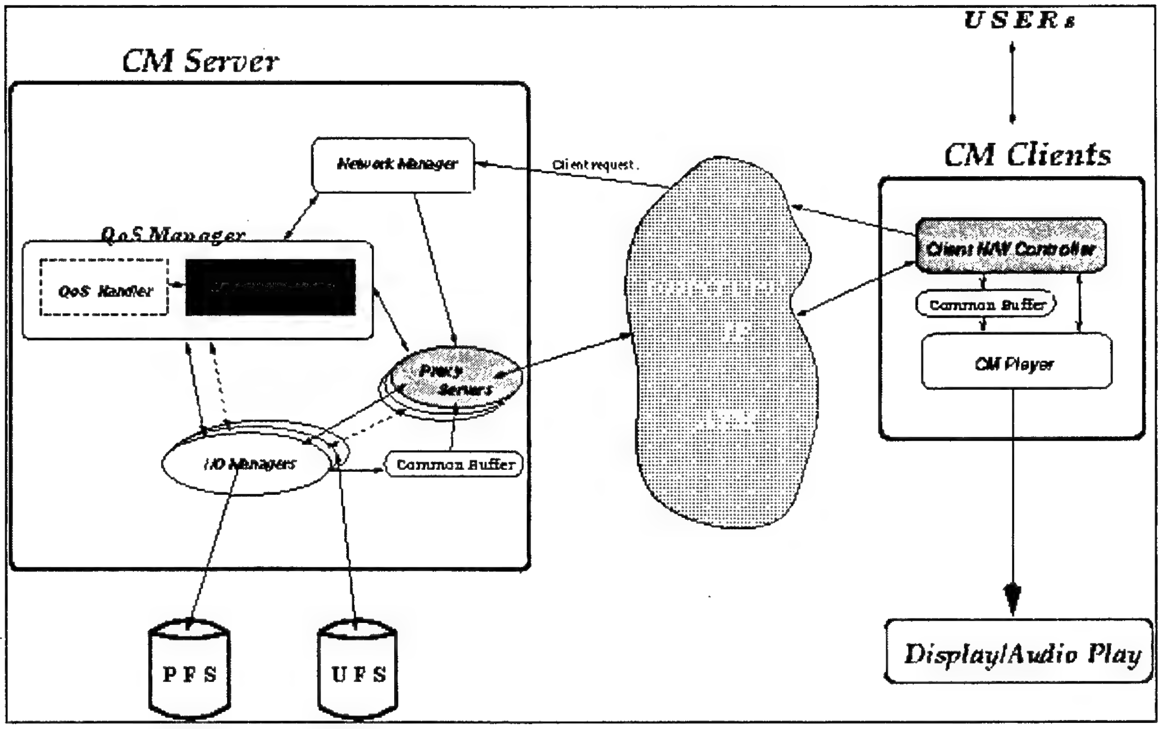


Figure 26. Architecture of Continuous Media Server

5.4.2 CM Client Architecture

The CM client requests stream-operations (such as open, play, pause, and close) to the CM server, and receives data from server—in some rate (given by client)—as well as displays the retrieved stream on the screen. It has two main modules: Client N/W Controller and CM Player.

The client N/W Controller communicates with CM server. It is responsible for forming requests for starting CM streams, changing playback rate and other QoS parameters, and stopping the connection. After the CM stream is started, this module keeps receiving data and put them into common buffers.

The client CM player periodically gets a logical data unit from the common buffers and plays it on the relevant display device.

5.4.3 Network Manager

When CM server is just started, only the network manager and the QoS manager exist. The network manager waits for a connection request from a client. It instantiates a proxy server upon receipt of a new connection request. The proxy server communicates with the client thereafter. After the proxy server runs, the network manager completes its job and will wait for another connection request. That is, the network manager delegates the connection management functionality to the proxy server once the latter is operational.

The client's request is in a generic structure that can be used for all types of streams, the structure contains elements such as stream name, data rate, sampling rate, client's PID, stream type (AU, MJPEG, etc.), and the preferred QoS parameters.

5.4.4 Proxy Server

5.4.4.1 Life Cycle of Proxy Server

One proxy server is related to each client connection. It is created when the connection is set up and dies when the connection is stopped. The proxy server is instantiated with the initial CM stream parameters by the network manager on a new client request. Right after it runs, the proxy server sends the client's initial CM stream parameters including stream name, stream type, data rate, QoS parameters, etc. to the QoS manager and waits for answer. The QoS manager checks if the CM stream request is admissible and answers the proxy server. If the request is rejected, the proxy server sends a message to the client and wait for a new request from the client.

If the request was accepted, the QoS manager creates an I/O manager to retrieve data from disk for the CM stream and returned a structure to the proxy server. The proxy server then sends a message to the client to acknowledge the CM stream request.

After the I/O manager starts the CM stream, the proxy server periodically sends LDUs to the client according to the data rate. The detailed policy for sending LDUs is discussed in the next subsection.

In addition to sending LDUs, the proxy server receives CM stream operation requests from the client. The operations include open-CM-stream, close-CM-stream, play, pause, fast-forward, set-rate, set-QoS-parameters and stop-connection. Upon each request, the proxy server forwards it to the QoS manager and waits for answer, and sends a message of acknowledgement or rejection to the client.

When the CM stream is finished, it is automatically closed. The proxy server sends a message to the client and waits for requests. Upon the stop-connection request, the proxy server notifies the QoS manager. Then the proxy server kills itself and finishes its life.

5.4.4.2 QoS Driven Dropping Mechanism

The major task of the proxy server is to send the CM stream to the client, and make it meet the preferred data rate and preferred QoS requirements.

The proxy server divides its service time into service cycles. The length of a service is decided by the playback rate of the CM stream. For instance, if the data rate is 30 frames/second, the service cycle is 1/30 second long. In the beginning of each cycle, the proxy server wakes up and sends out an LDU. Then it waits till the beginning of next service cycle. Once the CM stream begins, every service cycle is related with an LDU. An LDU is late for a service cycle if it is not ready at the beginning of the service cycle. In general cases, the proxy server wakes up on time and sends out the next LDU. However, there are some exceptions.

When a service cycle begins, the related LDU may not be ready. There are two reasons for this. One is that the LDU is late due to the uncertainties of the computer system. The other reason is that the LDU is scheduled not to be read out from the disk at all.

Another exception is that the proxy server may wake up late in a service cycle because the non-real-time operating system can not guarantee the required timing. We set a small period of time 't' according to the permissible drift. If the proxy server does not wake up till it 't' has elapsed in a service cycle, the proxy server is considered late.

We need to find a solution for the above cases. There are three approaches.

6. The first approach is to send the LDUs sequentially without any LDU dropping. This approach, which is called sequential mechanism, favors the ALF and CLF QoS parameters. Although the sequential mechanism has the best result for ALF and CLF, other QoS parameters may be very bad and the system's capability is restricted. First, if a LDU is late, the proxy server must wait till the next service cycle for it to be ready. This definitely results in LDU drift. Second, If the proxy server wakes up late and sends out the LDU sequentially, the LDU may reach the client late too. This may result in LDU drift. Third, since every LDU must be sent on the network, the network load is higher than that with LDU-dropping. This result in a higher probability for the network performance to degrade.
7. The second approach is called the pure dropping mechanism. When a LDU is late or the proxy server wakes up late, the proxy server drops the LDU and sends the next LDU instead. The pure dropping mechanism favorites the ADF and CDF QoS parameters. The drift factors gets the best results, but the LDU loss may increase to an unacceptable level.
8. The third approach tries to compromise between the loss factors and the drift factors, and is called QoS driven dropping mechanism. As stated in section 5.3.2, in order to achieve satisfiable video and audio effects, all the four QoS parameters must be lower than their lower bounds.

To keep the CLF less than 3, it is just sufficient to consider the following two cases:

case 1: O X X ? ?

case 2: O X O X ? ?

Here 'O' means LDU arrives on time and 'X' means LDU drops or lost. Then, what should we do for '?' slots? If either of them is 'X' then CLF must be bigger than 2, that is, we cannot guarantee the lower bound of CLF (= 2).

Therefore, we should send the '?' slots anyhow in these two cases even if they arrive somewhat late (due to sending, we may get some degradation on drift QoS factors (ADF/CDF)).

Under the QoS-driven dropping mechanism, we properly compromise the trade-off between the time (drift) and the loss QoS factors. Since the proxy server knows which LDUs were sent and which LDUs were dropped, it is easy to calculate LDUs ALF value and CLF value.

In this way, a LDU is dropped only when the dropping doesn't affect the video or audio effect, and the drift factors are kept as low as possible. Furthermore, in high load, some LDUs are not retrieved from the disks to save I/O bandwidth. Then these unretrieved LDUs must be dropped. The proxy server also knows which LDUs are not retrieved. So this QoS driven dropping mechanism helps to provide good performance by graceful degrading. We might have to think of more intelligence dropping strategies for a better QoS handling on video streaming.

5.4.5 CORBA Implementations

Extending the Socket interface to use CORBA requires some modifications to the original C/Socket code. We replaced all C socket calls with stubs and skeletons generated from a pair of CORBA interface definitions. One IDL interface (called CM_User) uses a sequence to transmit the data from server to client, and the other IDL interface (called CM_Request) has operations for opening a video stream from the server and six video functions such as play, fast-forward, slow-forward, pause, resume and stop. The video functions change the rate of playout with the client's process id and a given play rate.

The IDL interfaces used in the CM server system are as follows:

```
typedef sequence<octet> mjpg_frame;

interface CM_User {
  void putMJPGFrame(
    in mjpg_frame frame,
    in short length);
  };

interface CM_Request {
  oneway void playMJPG(
    in CM_User to_where,
    in string name,
    in short filesys,
    in short rate,
    in short clientpid,
    in short streamtype);
  oneway void ff(in short clientpid, in short newrate);
  oneway void sf(in short clientpid, in short newrate);
  oneway void play(in short clientpid);
  oneway void pause(in short clientpid);
  oneway void resume(in short clientpid);
  oneway void stop(in short clientpid);
  };
```

The putMJPGFrame operation of CM_User, which is part of the client interface, is called from the server (in proxy server) with two parameters: a sequence of MJPEG frame and its length. The playMJPG operations of CM_Request in the server side use oneway semantics for video functions since video distribution only needs unidirectional control data transfer (from client to server).

5.4.6 QoS Manager

Our QoS manager consists of admission controller and QoS handler. The admission controller is responsible for admission control for a CM stream request. Once the CM request is admitted, the QoS handler makes an I/O schedule for the CM stream.

5.4.6.1 Admission Controller

Every request from clients is sent to the admission controller. The admission controller checks if the request can be met without affecting the QoS of the existing CM streams. If the request can be met, then the request is passed to QoS handler. Otherwise, the admission controller tells the proxy server to reject the request.

A client's requests include open/close a CM stream, and rate control operations, such as play, pause, resume, fast-forward, set-rate, and stop-connection.

For the requests for playing, pausing, fast-forwarding, setting rate need the admission controller to check if the resource can meet the requirements.

5.4.6.2 Conditions of Admission Control

The admission controller provides two constraint tests: I/O bandwidth test and available buffer test. Each request (stream) arrives with some rate value (e.g. playback rate: i.e. LDUs per second). A request is admitted if it passes the two tests.

Suppose there are n CM streams in the server, to check the availability of disk I/O bandwidth for the new request in admission control, we use an I/O bandwidth test in Equation 1.

Equation 1: seek time + rotational Latency + transfer Time < cycleLength

From the equations we figure out that longer service cycle length helps to support more CM streams because lower percent of I/O time is wasted in seek latency and rotation latency. However, even if we could make the service cycle to be infinitely long, the number of CM stream we could support is limited. In fact, we can not make the service cycle infinitely long. The reason is that a longer service cycle requires a larger buffer. The available buffer size is bounded by the physical memory size. For a CM stream to be admitted, it must pass the available buffer test. Here we assume a dual buffer mechanism. The total buffer requirement must be smaller than available buffer size. Hence, the requirement of admission control is to satisfy equation I/O bandwidth test and buffer test. If either of them is not satisfied, the request must be rejected. When the acceptable ALF value for CM stream i is given as Pi, which means up to Pi percentage LDU loss is acceptable.

5.4.6.3 QoS Handler and Scheduling

All the requests admitted by the admission controller go to the QoS handler. The QoS handler takes care of data rate handling according to the input parameter of rate given by clients' requests. This module also controls the dynamic QoS negotiation and management.

1. For the open CM stream request, the QoS handler starts a new I/O manager for the stream and generates a disk I/O schedule for it.
2. For the close CM stream request, the QoS handler kills the relative I/O manager.
3. For the play/pause request, the QoS handler informs the I/O manager to start/stop disk I/O.
4. For fast-forward and set-rate, the QoS handler generate a new disk I/O schedule for the stream.

The major task for the QoS handler is to generate the disk I/O schedule for a CM stream. In generating disk I/O schedules, first the service cycle length must be properly determined by Equation 1. Then we decide the buffer size according to buffer test.

When the quality of service degrades in high load, some LDUs are scheduled not to be retrieved from the disk. We use the mechanism similar to the QoS driven dropping mechanism in proxy server to skip some LDUs. Since the two mechanisms are the same, the proxy server knows which LDUs are not retrieved from disk. Their cooperation helps to achieve performance gracefully degrading in high load.

When the client requests the stream to be played at the fast-forward mode, the schedule is changed to skip more LDUs.

5.4.7 I/O Manager

I/O manager handles the actual disk I/O and buffer management. The handler creates an I/O manager when a CM stream is opened. The I/O manager is killed when the stream is closed. When the I/O manager is created, it allocates the buffer according the buffer size decided by the schedule. When the I/O manager is killed, it releases its buffer. The job of an I/O manager is simple. The QoS handler has already decided the length of a service cycle, the size of its buffer, and which LDUs should be retrieved. The QoS handler wakes up in the beginning of each service cycle. It retrieves LDUs according to the schedule and put them into buffer for the proxy server to retrieve. Then the I/O manager sleeps till next service cycle.

5.5 Implementation Issues

Many implementation factors may affect the performance of our CM server. In this section, we list some of them, including the server model, file system, the number of threads, and packet size.

5.5.1 Server Model

Our CM server system is a typical client-server application. We have two choices as our server model: multi-threading or multiprocessing. It is not obvious what we should adopt between the two models. The biggest advantage of multi-threading is clear—context switch is less expensive. Moreover, CM server is an I/O bound type of application, multi-threading seems to be a good candidate since we can take advantage of thread concurrency with less cost in terms of context switch, memory requirement, etc. But, the time slot interval given to each process is limited, depending on CPU load at a particular time, it could take too long for a thread to get its turn and process a request or sending a frame. In addition to that, the streams are pretty independent if there is no admission control module existing; thus multiprocessing also seems to be a good candidate. Forking a new process to handle new request is straightforward, easy to program since there is no synchronization needed. This approach is taken in both CMT and Vosaic. Finally, we have decided to implement both of them and see how they perform. The result shows that multiprocessing (with expensive context switch and memory requirement) performs poorly comparing to multi-threading, which is more complicated to implement. The drift time and jitter time when server is running multiple processes are just too large to be tolerable by human perception. Hence, we use multi-threading for our CM server.

5.5.2 Number of Threads

In the CM server, every independent function unit should be a single thread. The network manager and the QoS manager exist as long as the CM server runs. Since the network manager manages the global client connection request, it should be a single thread. Because the QoS manager manages the admission control and generates the I/O schedule for each stream, it should know the requirements and status of each stream. So it must be a single thread.

The proxy servers and I/O managers do not exist unless there are some CM stream connections. Since there is a distinct connection between a client and the CM server, it is natural to create a proxy server to manage this connection. So, a proxy server should be a single thread. When we first implemented the I/O manager, it was in one thread because fewer threads would result in better performance. However, because of the synchronization between the QoS manager and the I/O managers, the synchronization between the proxy servers and the I/O managers made the I/O manager thread block frequently. This made the disk I/O performance really bad. Finally, we made an I/O manager for each CM stream and each I/O manager is a single thread.

5.5.3 File System

We have two choices of file system—P*resto* file system (PFS) [LSSKW97] and Unix File System (UFS).

PFS provides unit (a sequence of video frames or audio samples)-based retrieval mechanisms while UFS provides byte-oriented abstraction. In general, CM applications (such as an application that manipulate video streams) prefer unit-oriented abstraction. UFS is efficient in handling small size record-based file accesses, but is expensive for accessing larger files because of tree-structured index of the file system. Since the retrieval unit used by PFS manager can be so big (for example, 20 JPEG frames per access while 1 frame in UFS), the performance of PFS is far better than that of UFS.

5.5.4 Packet Size

How large a packet should be? CMT has chosen 8K, which is pretty arbitrary. The breaking of frames into packets and vice versa is necessary. UDP packets are limited in size; a whole full color 1024x768 frame will certainly be split out at the lower network level. Losing one fragment means losing the whole frame. If the CM delivery mechanism has some underlying network protocol, breaking large frames up into packets is reasonable since it allows retransmitting lost packets, not frames. But, we don't have any network protocol in mind yet, doing so would be redundant and would add extra work to both the client and the server. Consequently, sending frames is chosen instead of sending packets.

5.5.5 Integration with CORBA

We ported the first version of socket-based CM server on CORBA and got a CORBA-based CM server. The CORBA implementations were developed using single threaded versions of Orbix 2.0, which fully supported the OMG 2.0 CORBA standard. All C/socket calls were replaced with stubs and skeletons generated from a pair of CORBA interface definition language (IDL) specifications. The IDL specification uses sequence parameters for the data buffer rather than string parameters, which are a bit slow. It is because the IDL sequence mapping contains a length field, whereas the string mapping does not. This length field makes the IDL stubs not search each sequence parameter for a terminating NULL character.

The main drawback of using CORBA in CM Server is that the data copying overhead and the higher fixed overhead of the CORBA implementations considerably limits the performance.

For small buffer sizes, the fact stems from the higher fixed overhead of CORBA such as memory management makes the performance lower. For large buffer sizes, data copying overhead significantly affects the performance of CORBA-version and it limits the throughput.

Every time each request is invoked in CORBA-version, the request message of CORBA contains the name of its intended remote operation represented as a string. Thus CORBA demultiplexes incoming request messages to the appropriate up-call by performing a linear search through the list of operations in the IDL interface. Henceforth, operations in CORBA-version should be ordered by considering this fact (i.e. decreasing frequency of use).

5.5.6 Other Issues

The next issue involves the client side. The non-deterministic nature of best effort TCP/IP and UDP/IP makes it really hard to predict the delay time of each frame on the fly. With CM applications, late frames are considered lost frames, duplicates of UDP frames are even worse. The time-critical property of CM applications requires us to think of a way to prevent network delay and jitter as much as possible. The obvious way to do that is to have the client buffers data after some certain interval before playing back. This is exactly what is done with our CM clients.

There is one point worth noticing here, which is that at both the client and the server, there is a classical computer science problem: the producer/consumer problem. At the client side, the module that read frames through network and puts them into some common buffer is the producer, while the CM playback module is the consumer. On the server side, the module that takes frames from buffer and sends them to the client is the consumer, and the module that reads data from disk, breaks them down to frames then puts them into the buffer is the producer. Here, we use condition variables and mutexes to solve the problem.

5.6 Conclusions

We have attempted to summarize some of the CM server architecture and techniques to implement distributed CM applications. We have specified and represented QoS metrics that is applicable to lossy channel like UDP and done our CM server performance evaluation based on those metrics. We have described the design issues, implementation details of CM servers working for both C/Sockets and CORBA versions.

Our on-going work includes developing more intelligent algorithms and mechanisms of admission control and QoS manager that take advantage of QoS specifications to optimize system performance. Especially, we can take advantage of the fact that even UDP is a lossy protocol, the number of frames lost is not that many, and in most cases, is still in an acceptable range. This fact gives us an important observation that adding some control channel, having more intelligent dropping mechanisms will give us good performance in terms of both timing quality and number of lost frames. We expect good performance results using the QoS-driven dropping mechanism on timing drift and loss QoS factors.

Section 6
Demonstration Application: Planning and Monitoring Air Combat

DARPA is currently conducting the Joint Force Air Component Commander (JFACC) program to develop information technology for air operations planning and monitoring. The overall JFACC objective is to make the planning process continuous, objectives-based, and integrated across all resources. This can only be done if there is an explicit, comprehensive computer representation of the air operations plan and supporting situation data, shared by all collaborators in the planning process.

The HDIMI project addresses five key information management problems relevant to JFACC:

- **Meeting applications' information needs**—Various JFACC applications and their human operators require different information at different levels of abstraction. They should be provided the information they need, but no more, through views of the air operations plan as shown in Figure 27. As an example, AOC-level planners schedule specific tasks and assign them to units for execution. These planners must be able to define and view all tasks and their assignments. At the wing and unit levels, planners fill in finer details such as assignment of specific personnel and aircraft. In general, the information needs of AOC-level planners are broader in scope but lesser in detail than those of wing and unit-level planners.

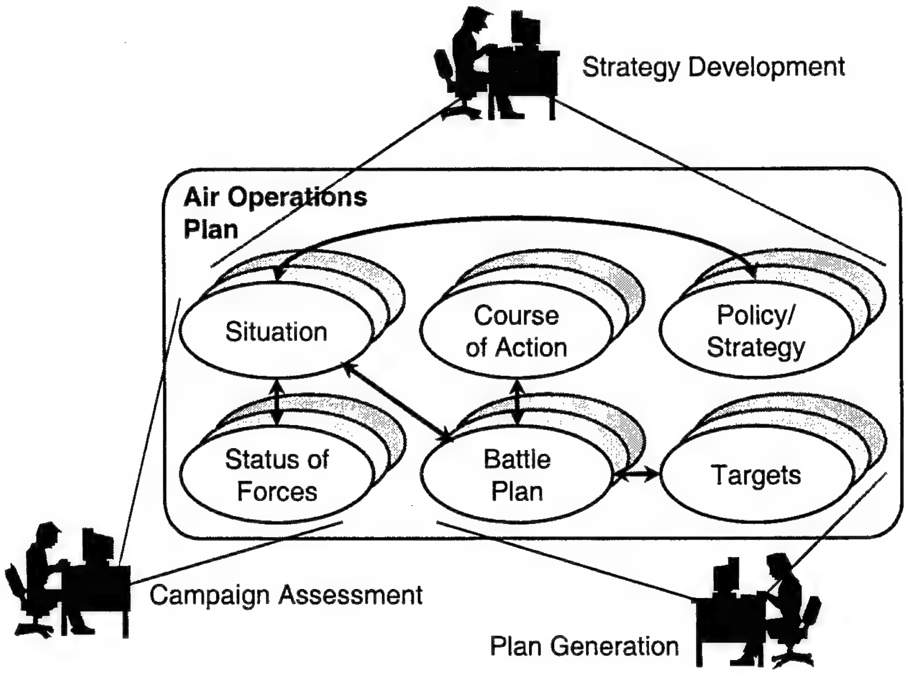


Figure 27. Different applications require different views of plan and situation data.

- **Support for multimedia data**—Planners need to be able to view multimedia situation data such as maps and imagery of potential flight paths and target zones, annotated with weather, enemy forces and other entities of interest. The information management infrastructure must therefore handle multimedia data types as well as conventional structured data.

- **Notifying applications of plan and situation changes**—Plans are typically based on assumptions about the situation, our objectives and resources, and estimates of theirs. Planning applications must be notified when these assumptions no longer hold. Similarly, situation displays should be updated dynamically as new information is recorded in the situation database. However, applications must not be burdened with notifications of changes to data they don't care about.
- **Access to historical information**—Databases typically hold information about the current state of the world: the current plans, current situation, etc. The information management system should provide simple, uniform access to historical information for trend and pattern analysis, review of the decision-making process, training, and other purposes.
- **Simulation and "what if"**—Effective planning in a dynamic battlefield situation requires the ability to predict future situations based on the current situation, our plans, and projections of enemy actions. Simulation is a category of applications unto itself, not part of the information management function. However, the information management system should encapsulate simulators so that their output can be accessed seamlessly using the same services by which historical information is accessed.

As part of the HDIMI project, we have developed a demonstration application called "Planning and Monitoring Air Combat" (PMAC) that illustrates solutions to the first three items. The last two relate to the Active Views history abstraction, which has been defined but not implemented in Sonata.

This section describes the PMAC application. Included are:

- An application overview;
- The user interfaces;
- The ObjectStore schema, originally defined using Booch notation with Rational Rose;
- The transactions for initializing the database and for conducting the scenario;
- A scenario in terms of those transactions.

6.1 Application Overview

The demonstration involves three types of operators: an Air Operations Center (AOC) planner, one or more Wing Operations Center (WOC) planners, and a simulation operator. The AOC planner performs the following functions:

- Generates target objectives and desired complement of platforms to be used for the strike.
- Assigns each target objective to a particular wing on the basis of availability of the required platform types at the time of the strike.
- Reassigns a target objective if the originally assigned wing fails to achieve the objective.

The WOC planner performs the following functions:

- Assigns specific platforms of the required types to target objectives.
- Revises assignments of platforms to objective as necessary, e.g. if an assigned platform becomes non-operational. If an objective becomes infeasible for the wing to accomplish given available platforms, the WOC planner can mark the objective as "failed", thereby handing the objective back to the AOC planner for reassignment.

The simulation operator changes the status of individual air platforms from "FullyOperational" to "PartlyOperational" or "NonOperational", representing failures of some sort that prevent participation in a mission.

6.2 User Interfaces

The AOC planner, WOC planner, and simulation operator each have a dialog box in which they can enter transactions to update the database. See Section 0 for the list of transactions. In addition, the AOC planner has an Active View of all TargetObjectives and all associated attributes, displayed in tabular format; the WOC planner has a similar display of the TargetObjectives assigned to his wing. Each also has a map-oriented display of target objectives with color coding to indicate whether the objective has aircraft of sufficient number and correct type assigned to strike it.

6.3 Schema

The schema was developed using Rational Rose. It includes class definitions and accompanying documentation. Figure 28 is the Booch diagram for the schema. Class specifications exported from Rational Rose follow.

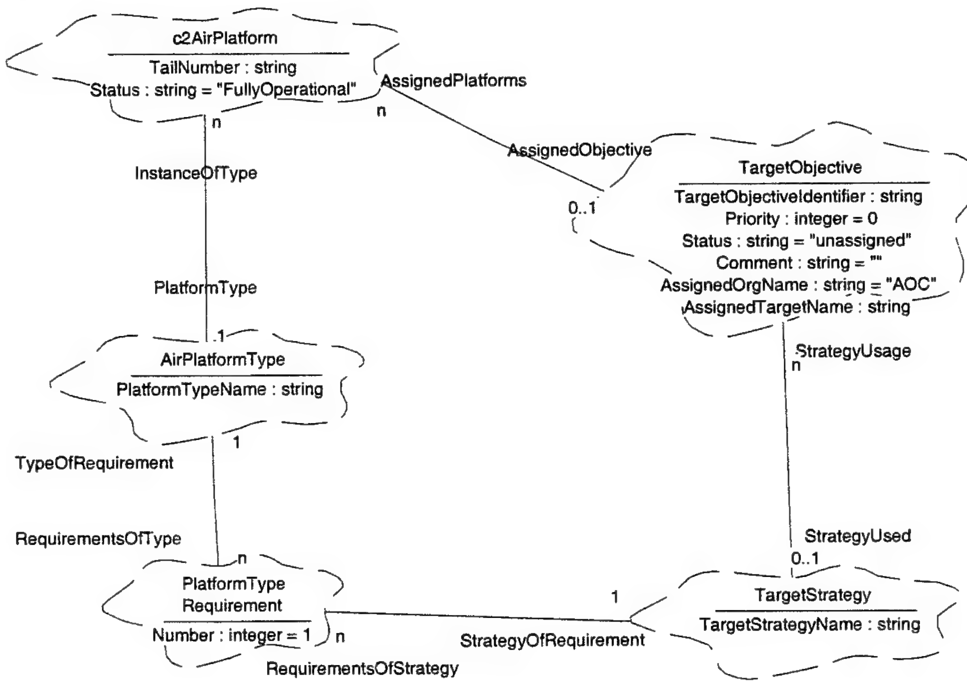


Figure 28. PMAC Application Schema

Class name:
   c2AirPlatform

```
Category: <Top Level>
Documentation:
   An entity whose primary operating environment is
   between 0 and 100,000 feet above sea level

   Actual location and status are updated by the
   simulation operator for the purpose of demonstration.
   Speed and heading are inferred from these.

Export Control: Public
Cardinality: n
Hierarchy:
   Superclasses: none
Associations:

         AssignedObjective : TargetObjective
         PlatformType : AirPlatformType

Public Interface:
   Attributes:
                 TailNumber : string
                         Unique identifier of the aircraft

                 Status : string = "FullyOperational"
                         This attribute is in the C2
                         schema w/  no documentation. We
                         will take it to mean operational
                         status: ability to perform its
                         function: "FullyOperational",
                         "PartlyOperational"
                         "NonOperational"
```

Class name:
   AirPlatformType

```
Category: <Top Level>
Documentation:
   Introduced by JPR. Models a model of air platform (e.g.
   F15) as opposed to a specific instance of it.

   Instances are created manually in the DB for the
   purposes of demonstration.

Export Control: Public
Cardinality: n
Hierarchy:
   Superclasses: none
```

```
Associations:

          InstanceOfType : c2AirPlatform
          RequirementsOfType : PlatformTypeRequirement

Public Interface:
   Attributes:
                PlatformTypeName : string
```

Class name:
   TargetObjective

```
Category: <Top Level>
Documentation:
   (This model departs from the JTF model) An instance of
   TargetObjective represents a need to strike a
   particular target in a particular timeframe. There
   could be multiple TargetObjectives for the same target.

   Created by AOC Planner for purposes of demonstration.
   Modified by WOC Planner.

Export Control: Public
Cardinality: n
Hierarchy:
   Superclasses: none
Associations:

          AssignedPlatforms : c2AirPlatform
          StrategyUsed : TargetStrategy

Public Interface:
   Attributes:
                TargetObjectiveIdentifier : string
                Priority : integer = 0
                Status : string = "unassigned"
                        Status of the TargetObjective.
                        Values include:
                        Unassigned: WOC has not assigned
                        the objective to a wing to
                        accomplish
                        Assigned: WOC has assigned the
                        objective to a wing, but the wing
                        has not allocated platforms and
                        designated a start and end time
                        Scheduled: WOC has allocated
                        platforms and designated a start
                        and end time
                        Failed: Wing cannot achieve the
                        objective as specified by the AOC
```

```
                    (Put reason why not in Comment
                    field)
                    Achieved: Wing has accomplished
                    the objective

               Comment : string = ""
                    Text field for communication
                    between AOC and WOC, e.g. if
                    Status is Failed, reason for
                    failure

               AssignedOrgName : string = "AOC"
                    The name of the organization
                    currently responsible for
                    achieving the objective

               AssignedTargetName : string
                    Printable name of a target that
                    this objective is directed
                    against
```

Class name:
PlatformTypeRequirement

```
Category: <Top Level>
Documentation:
   Defines the number of instances of a particular
   PlatformType needed for a particular TargetStrategy

   Instances are created manually in the DB for the
   purposes of demonstration.

   This object type could be replaced by an n-n
   association between AirPlatformType and TargetStrategy,
   with an associated Number attribute.

Export Control: Public
Cardinality: n
Hierarchy:
   Superclasses: none
Associations:

         StrategyOfRequirement : TargetStrategy
         TypeOfRequirement : AirPlatformType

Public Interface:
   Attributes:
              Number : integer = 1
                    The number of instances of a
                    given PlatformType required for
```

```
                              the given TargetStrategy
```

Class name:
   TargetStrategy

```
Category: <Top Level>
Documentation:
   A particular strategy for attacking targets.

   Instances are created manually in the DB for the
   purposes of demonstration.

Export Control: Public
Cardinality: n
Hierarchy:
   Superclasses: none
Associations:

         StrategyUsage : TargetObjective
         RequirementsOfStrategy: PlatformTypeRequirement

Public Interface:
   Attributes:
                TargetStrategyName : string
                        Identifies a particular strategy
                        for attacking targets
```

6.4 Transactions

The transactions have a command-line syntax consisting of a command name followed by zero or more parameters. Some parameters are optional, as indicated below. A command line syntax is used so that transaction script files can be written and executed.

The transaction names are meant to be short but reasonably mnemonic. The following are transaction naming conventions:

- The name of a transaction that creates an object is an abbreviated form of the object's class name.
- The name of a transaction that deletes an object is the same name, preceded by "x".
- Queries begin with "q".

6.4.1 AirPlatformType

```
apt PlatformTypeName
```

Creates an `AirPlatformType` with the specified `PlatformTypeName` (key attribute).

```
xapt PlatformTypeName
```

Deletes an `AirPlatformType` and any associated instances of `PlatformTypeRequirement`. Fails if there are any associated instances of `c2AirPlatform`.

```
qapt
```

List all `AirPlatformTypes`.

6.4.2 TargetStrategy

```
ts TargetStrategyName
```

Creates a `TargetStrategy` with the specified `TargetStrategyName` (key attribute).

```
xts TargetStrategyName
```

Deletes a `TargetStrategy` and any associated instances of `PlatformTypeRequirement`.

```
tsrqt TargetStrategyName PlatformTypeName Number
```

Specifies the required number of a platform type for a given target strategy. Both the type and strategy must exist. If `Number` <=0, the requirement is removed.

```
qts
```

Lists all `TargetStrategys`, including the required number of each `AirPlatformType`.

6.4.3 TargetObjective

```
to TargetObjectiveIdentifier AssignedTargetName Priority
```

Creates a `TargetObjective` with the specified `TargetObjectiveIdentifier` (key attribute), `AssignedTargetName`, and Priority. Sets `Status=`"`Unassigned`", `Comment=`"", `AssignedOrgName=`"", and `StrategyUsed=nil`.

```
xto TargetObjectiveIdentifier
```

Deletes `TargetObjective` and unassigns any platforms assigned to that objective.

```
tostrat TargetObjectiveIdentifier [TargetStrategyName]
```

Assigns the specified `TargetStrategy` (or `nil`) to the specified `TargetObjective`.

```
toorg TargetObjectiveIdentifier [AssignedOrgName]
```

Assigns the specified `AssignedOrgName` to the specified `TargetObjective` and sets its Status to "`Assigned`". If no `AssignedOrgName` is given, set `AssignedOrgName` to "" and Status to "`Unassigned`".

```
qto [AssignedOrgName]
```

Lists all `TargetObjectives`, with all attributes and `TargetStrategyName` if any. If `AssignedOrgName` is specified, list all that are assigned to the specified organization.

```
qtoap [TargetObjectiveIdentifier]
```

Lists the specified `TargetObjective`, with all attributes, the `TargetStrategyName`, if any, and the `TailNumbers` assigned to that `TargetObjective`, grouped by `PlatformTypeName`. If `TargetObjectiveIndentifier` is not specified, do this for all `TargetObjectives`.

6.4.4 AirPlatform

```
ap PlatformTypeName TailNumber OwnerOrgName
```

Creates a `c2AirPlatform` of a specified `AirPlatformType`, `TailNumber` (key attribute), and `OwnerOrgName`. Sets `Status` to "`FullyOperational`" and `AssignedObjective` to `nil`.

```
xap TailNumber
```

Deletes a `c2AirPlatform` and any associations it has.

```
apstat TailNumber Status
```

Sets the `Status` of the specified `c2AirPlatform`.

```
apobj TailNumber [TargetObjectiveIdentifier]
```

Assigns a `c2AirPlatform` to the specified `TargetObjective` or unassigns it.

```
qap [OwnerOrgName]
```

List all `c2AirPlatforms`, with all attributes, associated `PlatformTypeName`, and assigned `TargetObjectiveIdentifier`, if any. If `OwnerOrgName` is specified, list all that are owned by the specified organization.

6.5 Scenario

This section defines a sample scenario to illustrate the Active View mechanism. Much more can be added to show how various database updates influence the Active Views; this is just a start.

Also, the definitions of the target strategies and the appropriateness of them for particular target objectives needs to be reviewed and corrected by an air operations planning domain expert.

The scenario has two parts: database initialization and the run-time scenario. The database initialization need only be done once. The run-time scenario can be done multiple times. In the scenarios, actual transactions are shown in plain text; comments about what is about to be done and what the results should be are in *italics.*

6.5.1 Database Initialization

The following is database initialization, performed before the scenario takes place. It includes defining the air platform types, actual instances of air platforms, and target strategies.

Initialization Transaction
apt A-10
apt A-4
apt A-6
apt A-7
apt B-2
apt B-52
apt F-104
apt F-111
apt F-117
apt F-15
apt F-16
apt F-5
apt FA-18
ap F-5 F-5.1 Wing1
ap F-5 F-5.2 Wing1
ap F-5 F-5.3 Wing1
ap F-5 F-5.4 Wing1
ap F-5 F-5.5 Wing1
ap B-2 B-2.2 Wing1
ap B-2 B-2.2 Wing1
ap F-117 F-117.1 Wing1
ap F-117 F-117.2 Wing1
ap F-117 F-117.3 Wing1
ts DayCloseAirSupport
tsrqt DayCloseAirSupport A-10 4

Initialization Transaction
ts DaySmallTarget
tsrqt DaySmallTarget F-5 4
ts StealthBigTarget
tsrqt StealthBigTarget B-2 3
tsrqt StealthBigTarget F-117 4
ts NightSmallTarget
tsrqt NightSmallTarget FA-18 2
tsrqt NightSmallTarget F-16 2

6.5.2 Run-Time Scenario

The following is a simple run-time scenario. It can be easily extended to include more transactions or to add another WOC. The scenario is divided into three columns to indicate what happens on the three user interfaces.

AOC	Wing 1 WOC	Simulation Operator
Query air platform types.		
qpt		
Result should be the types entered during initialization		
Query target strategies.		
qts		
Result should be list of target strategy requirements entered during initialization.		
Create target objectives and define strategies.		
to t1 PowerPlant 1		
to t2 CommandPost 3		
tostrat t1 DaySmallTarget		
tostrat t2 StealthBigTarget		
Assign target objectives to wing.		
toorg t1 Wing1		
	Active view of target objectives should show the newly assigned target objective.	
	Find out what aircraft the wing has	

AOC	Wing 1 WOC	Simulation Operator
	and their status qap Wing1 *Result should be the five F-5's entered earlier. Now assign four of them to the TargetObjective* apobj t1 F-5.1 apobj t1 F-5.2 apobj t1 F-15.3 apobj t1 F-15.4 *With the assignment of the fourth F-5 to the target objective, the WOC active should be updated to show that TargetObjective t1now has sufficient aircraft to satisfy its objectives*	
		Mark one of the assigned aircraft as nonoperational apstat F-5.3 NonOperational
	The WOC Active View should show that TargetObjective t1 no longer has sufficient aircraft, since one of the assigned ones is NonOperational. Now query the set of aircraft again to find out which one went NonOperational and to locate an Operational one that is not assigned qap Wing1 *Result should show five F-5's belonging to Wing1, with F-5.3 NonOperational and assigned, but F-5.5 Operational and unassigned. Fix the assignment.* apobj F-5.3 apobj F-5.5 t1 *The WOC Active View should show that TargetObjectivet1 has sufficient aircraft again.*	

Section 7
Conclusions

The HDIMI project has substantially achieved its technical objectives, as summarized in Section 1.3. While we have not been able to achieve all the goals we set out at the beginning of the project (distributed resource management, content-based query), we believe that Sonata represents a significant technical advance in active data management.

7.1 Dual-Use Applications

We believe that Active Views and the related technologies developed in this project have significant value to military command and control systems, Honeywell's monitoring and control system offerings, and the broader information technology market—any domain where "situation awareness" is a key application requirement.

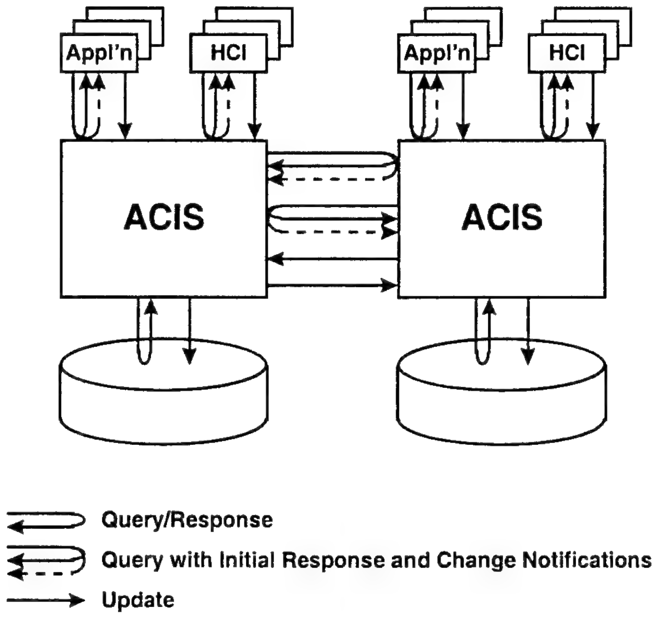
Honeywell is implementing a strategy to commercialize this technology through third-party vendors that serve this broader market. We believe that this form of commercialization is in the best interests of both Honeywell and the Government. As an incentive to potential commercializers, Honeywell has filed a patent application on key Active View concepts and would license the patent to third-party commercializers.

7.2 Recommendations for Future Work

The HDIMI project has focussed primarily on developing broadly-applicable information management technology as opposed to C^4I applications. The technology is sufficiently mature that it should be used and evaluated in the context of application-oriented programs such as DARPA's JFACC, Dynamic Database, and Adaptive Information Control Environment (AICE) programs. To this end, Honeywell and the University of Minnesota have submitted a number of proposals to DARPA. All have been deemed "selectable" but none has been funded. We continue to pursue this course of action and solicit AFRL assistance.

A second path that can be pursued simultaneously is to extend the capabilities of the existing Sonata technology. The four projects listed below represent our recommendations in this direction.

7.2.1 Active Collaboration Information Server (ACIS)



Objective: Develop an *Active Collaboration Information Server* (*ACIS*) that supports large-scale collaboration and situation awareness applications such as JFACC and Command Post of the Future. ACIS is an active, distributed information server that provides applications and users with consistent, dynamically updated information that meets their specific collaboration and situation awareness needs. ACIS supports structured data and multimedia data types such as images, text, and video to provide realistic depictions of the situation.

Approach: Build on Active Views technology demonstrated in the HDIMI project. Active views are an integrated set of mechanisms that provide uniform treatment of state, state change, and state history for arbitrary object types throughout a system. These objects include sensors and user interface objects as well as database-resident objects. Enhance the technology to provide a robust, scalable, software capability that makes effective use of COTS technology and that can be readily integrated in large-scale technology demonstrations. Extensions to improve functionality, scalability, performance, and reliability include:

- Distributed active servers with selective information replication and distribution;
- Dynamic Active View creation to satisfy new information needs while the system is running;
- Incremental view maintenance to reduce processing and bandwidth requirements;
- View materialization to improve response time;
- Aggregation of similar views to reduce redundant computation.

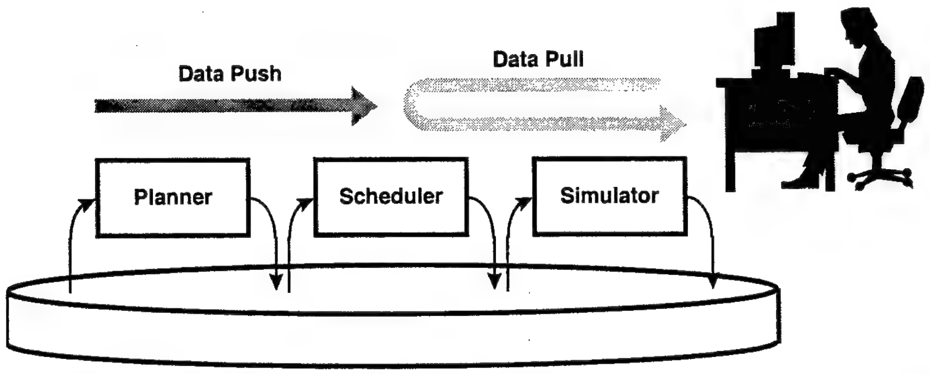
Demonstrate and evaluate ACIS as a basis for large-scale collaboration through a technology integration experiment with selected C^4I applications.

Capability Developed: An active, distributed information server implementing Active Views technology using COTS DBMS and ORB.

Potential Users: C^4I applications requiring information-intensive, distributed collaboration or situation awareness, e.g. JFACC, Command Post of the Future.

7.2.2 ACIS-Application Integrated Environment (ACIS-AIE)

Objective: Extend ACIS to be a general-purpose C^4I decision support environment that transparently integrates data storage/retrieval and application-specific computation in response to user queries. For example, to answer the query "*What are our options for evacuating civilians and what are their likely outcomes?*", ACIS will invoke planning, scheduling, and simulation applications, initializing them from stored battlefield situation data and storing intermediate results back in the database. As the figure illustrates, ACIS-AIE uses both data push and data pull. Some applications may run prior to a user's query (data push) and some in response to it (data pull). A client should be able to pose queries and perceive no difference, except possibly in response time, between these two modes.



Approach: Providing such a capability requires several technical advances, including:

- Well-defined semantics for referencing large-granularity application-specific functions in a query;
- Seamless integration of database and simulation techniques to answer queries about the present and future situations;
- Optimization techniques that generate efficient plans for computing query results, interleaving database access and application execution;
- Techniques that allow combinations of data push and data pull, invisibly to clients.

Capability Developed: An active, distributed decision support environment that integrates C^4I applications and database access to meet users' information needs.

Potential Users: C^4I applications involving situation awareness, planning, and simulation.

7.2.3 Quality-Adaptive Information Management (QAIM)

Objective: Develop and demonstrate information management functions that adapt their information quality in response to user requirements and system conditions. Here, quality is defined at the information management level, and includes accuracy of the information provided, as well as its timeliness and precision. Embed the techniques in the evolving Active Collaboration Information Server.

The quality of decision making in a CPOF-type command and control scenario is critically dependent on the quality of information available to the decision-makers. Specifically, it is important to ensure that the *current state* (captured by the current values of the state variables) observed by the decision-makers is close enough to the *current real-world situation*. The current state includes incoming audio and video, various data displays, and results of computations.

Providing high quality information with low cost is of great interest in the command and control environment, e.g. shared situational awareness requirements for the CPOF. Ongoing DoD sponsored efforts are addressing some aspects of the problem, e.g. DARPA's program on quality of service based resource management (Quorum). However, so far almost all effort on information quality has been in the context of networks and resource management, and especially for continuous media. From a command-and-control view point, one could reasonably argue that the underlying infrastructure must provide its best-effort for information provisioning, and for data types such as audio and video this is quite acceptable. However, for many other kinds of data, e.g. computed values, sensor readings, etc., it is important to make the information quality explicit, so that the application knows how much it can rely on the information for its decision making. This requires that the ideas of quality of information and its servicing be an integral and explicit part of the information management capability.

Approach: Our approach will extend the concept of information quality to various types of information, including numeric and aggregate values, computed values, sensor readings, images, etc., in addition to audio and video. We will develop models for information quality and mechanisms for delivering quality information, and incorporate them in the ACIS system. Technical innovations include:

- Models of information quality for numeric and aggregate values, computed values, etc.
- Mechanisms, i.e. algorithms, for providing information with requisite quality.
- Analysis of information quality vs cost tradeoffs.

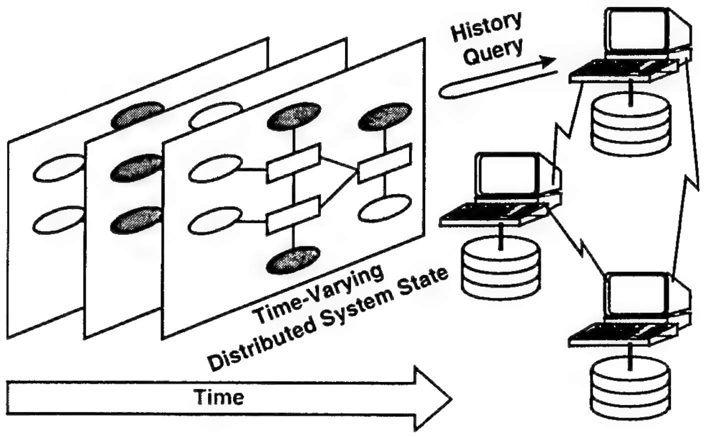
Many of these techniques affect the core functionality of COTS DBMS products. We will pursue approaches that work around the limitations of COTS products.

Capability Developed: The capability developed will be extensions to the Active View definition language to specify the information quality requirements of various kinds of data types. This specification will be used by the underlying information management mechanisms for information provisioning with the desired quality of service.

Potential Users: C^{4}I applications involving situation awareness.

7.2.4 Temporal ACIS

Objective: Extend ACIS to provide robust support for the time dimension in distributed C^{4}I information management systems. For example, the query "*Summarize the changes made to the air operations plan over the last shift*" requires access to potentially large volumes of historical data on AOC and WOC systems. The query "*How were our aircraft deployed at the time of the attack?*" requires temporal alignment of the historical information at multiple sites and a representation of the uncertainty of the sequence of events.



Approach: Extend Active Views history concepts to encompass histories of complex, distributed collections of objects. Key technical issues to be addressed include:

- A precise definition of summaries or views of object state *changes* (as opposed to database states).
- Imperfect information about the relative ordering of events in a distributed system.
- Efficient searching of object histories for large numbers of arbitrary objects (as opposed to "conventional" temporal relational data) in a distributed system.

Potential Users: Distributed C^4I applications involving historical search and analysis.

"*Those who cannot remember the past are condemned to repeat it.*"—George Santayana

"*To be a successful soldier you must know history.*"—Gen. George S. Patton

Section 8
References

[AKPBV96] Agrawal, M., Kenchammana-Hosekote, D. R., Pavan, A., Bhattacharya, S., and Vaidyanathan, N. *High Performance Network Services for Multimedia-Integrated Distributed Control.* Technical report, Honeywell Technology Center, Minneapolis, MN, July 1996.

[AU72] Aho, A.V and Ullman, J. D. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, 1972.

[BM93] Babaoglu, O. and Marzullo, K. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. *Distributed Systems*, S. Mullender, ed., Addison-Wesley, 1993.

[CATT97] Cattell, R.G. et al. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[CHAP96] Chappell, D. *Understanding ActiveX and OLE.* Microsoft Press, 1996.

[CJCS98] Chairman of the Joint Chiefs of Staff, *Joint Vision 2010*, 1998. http://www.dtic.mil/doctrine/jv2010/

[CKAK94] Chakravarthy, S., Krishnaprasad, V., Anwar, E., and Kim S.-K. Composite Events for Active Databases: Semantics, Contexts and Detection. *Proc. 20th Int'l. Conf. on Very Large Data Bases*, Santiago de Chile, Chile, Sept. 1994. pp. 606-617.

[CTC] Zhigang Chen, See-moong Tan, Roy H. Campbell, and Yongcheng Li, *Real Time Video and Audio in the WWW.*

[DGG95] Dittrich, K. R., Gatziu, S., and Geppert, A., eds. The Active Database Management Manifesto: A Rulebase of ADBMS Features. *Proc. Rules in Database Systems (RIDS '95)*, Athens, Greece, Sept. 1995, pp. 3-20.

[FLAN97] Flannagan, D. *Java in a Nutshell, 2nd ed.* O'Reilly and Associates, 1997.

[FS97] Fayad, M., and Schmidt, D. Object-Oriented Application Frameworks. *Communications of the ACM*, Vol. 40, No. 10 (October 1997) 32–38.

[GH93] Guttag, J. V. and Horning, J. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[GHJV95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, 1995.

[GM95] Gupta, A. and Mumick, I. S. Maintenance of Materialized Views: Problems, Techniques, and Applications. *Data Engineering Bulletin*, June 1995.

[GR93] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[GRAY96] Gray, J. et al. The dangers of replication and a solution. *Proc. 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, June 1996.

[GYT96] Gosling, J., Yellin, F., and the Java Team. *The Java Programming Language.* Addison-Wesley, 1996.

[HAEBE88] Haeberti, P.E. ConMan: A Visual Programming Language for Interactive Graphics, *Computer Graphics*, August 1988.

[HJHM+97] Huang, J., Jha, R., Heimerdinger, W., Muhammad, M, Lauzac, S., Kannikeswaran, B., Schwan, K., Zhao W., and Bettati, R. RT-ARM: A real-time adaptive resource management system for distributed mission-critical applications. *Workshop on Middleware for Distributed Real-Time Systems*, San Francisco, RTSS-97.

[INGAL88] Ingalls, D., et al. Fabrik: A Visual Programming Environment, Object-Oriented Programming: Systems, Languages, and Applications, *Special Issue of ACM SIGPLAN Notices*, Vol. 23, No. 11, November, 1988

[J6I93] C4 Architecture & Integration Division (J6I), J6, The Joint Staff, *C^4I for the Warrior*, June 12, 1993.

[KASS92] Kass, M. CONDOR: Constraint-Based Dataflow, *Computer Graphics.* July, 1992.

[KR94] Kilov, H. and Ross, J. *Information Modeling: An Object-Oriented Approach.* Prentice-Hall, 1994.

[KR98] Kuno, H.A. and Rundensteiner, E. A. Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation. *IEEE Transactions on Knowledge And Data Engineering 10*, 5 (September/October 1998), 768-792.

[LSNS98] Lee W., Su, D, Ngo, H.Q., and Srivastava, J. A QoS-Driven Networked Continuous Media Server. *SPIE International Symposium on Lasers, Optpelectronics, and Microphonics (Electronic Imaging and Multimedia Systems: SPIE Symposium on Photonics China—PC '98)*, Beijing, China, September 1998.

[LSSKW97] Lee, W., Su. D, Srivastava, J., Kenchammana-Hosekote, D.R., and Wijesekera, D., *Experimental Evaluation of PFS Continuous Media File System,* ACM International Conference on Information and knowledge Management (CIKM 97), Nov. 1997.

[LW94] Liskov, B. H. and Wing, J. M. A behavioral notion of subtyping. *ACM TOPLAS 16*, 6 (November 1994), 1811-1874.

[MEYE90] Meyer, B. *Introduction to the Theory of Programming Languages*, Prentice-Hall, 1990.

[MEYE97] Meyer, B. *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.

[OMG] Object Management Group, *CORBAservices: Common Object Services Specification*, http://www.omg.org/library/csindx.html.

[OMG95] Object Management Group. *The Common Object Request Broker: Architecture and Specification. Revision 2.0.* July, 1995.

[OUSTE93] Ousterhout, J. *Tcl and Tk Toolkit*. Addison-Wesley, 1993.

[PATE93] Paton, N. W. et al. Dimensions of Active Behavior. *Proc. Rules in Database Systems (RIDS '93)*, Edinburgh, Scotland, Sept. 1993, pp. 41-57.

[PATEL95] Patel, K. *Introduction to the Continuous Media Toolkit (CMT)*. Berkeley Multimedia Research Center, 1995.

[PS97] Pazandak, P., and Srivastava, J. Evaluating Object DBMSs for Multimedia. *IEEE Multimedia*, 4, 3 (July-September 1997) 34–49. Also submitted to AFRL as a COTS technology evaluation report under the HDIMI contract.

[SF97] Schmidt, D., and Fayad, M. Lessons Learned: Building Reusable OO Frameworks for Distributed Software. *Communications of the ACM*, Vol. 40, No. 10 (October 1997) 85–87.

[SL94] Stepanov, A. and Lee, M. The Standard Template Library. Technical Report HPL-94-34, Hewlett-Packard Laboratories, April 1994.

[SMITH94] Smith, B.C., *Implementation Techniques for Continuous Media Systems and Applications,* PhD thesis, University of California, Berkeley, Computer Science Department, 1994.

[SPIV98] Spivey, J. M. The Z Notation: A Reference Manual, 2nd ed., 1998. Published by the author at http://spivey.oriel.ox.ac.uk/~mike/zrm/.

[STE96] Steinmetz, R. Human Perception of Jitter and Media Synchronization, *IEEE Journal on Selected Areas in Communication*, Vol. 14, No. 1, pp. 61-72, 1996.

[SVK93] D.B. Stewart, R.A. Volpe, and P.K. Khosla. *Design of Dynamically Reconfigurable Real-Time Software using Port-based Objects.* Technical Report CMR-RI-TR-93-11, Department of ECE, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, July 1993.

[TAC+95] David Tennenhouse, Joel Adam, David Carver, Henry Houh, Michael Ismert, Christopher Lindblad, William Stasior, David Wetherall, David Bacher, and Theresa Chang. The ViewStation: A Software-Intensive Approach to Media Processing and Distribution. *Multimedia Systems* Vol 3, pp. 104-115, 1995.

[TANS93] Tansel, A. U., et al., eds. *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, 1993.

[TFS94] Teknowledge Federal Systems, *Joint Task Force Architecture Specification*, April 13, 1994. Updated information available at http://jtfweb3.nosc.mil/.

[TG95] Thompson, J. and Gottlieb. *Macromedia Director Developers Guide to Lingo*, 1995.

[TK77] Tsichritzis, D., and Klug, A. *The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Data Base Management Systems*, AFIPS Press, 1977.

[USER98] *Sonata User and Administration Manual*, submitted to AFRL as an HDIMI project deliverable, 1998.

[WC96] Widom, J. and Ceri, S., eds. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[WS96] Wijesekera, D, and Srivastava, J. Quality of Service (QoS) Metrics for Continuous Media, *Multimedia Tools and Applications*, Vol 3, No. 1, pp. 127-166, September. 1996.

Section 9
Publications and Patents

9.1 Publications

This list includes papers published since the end of the Multimedia Database Management System project that were funded by Rome Lab, based on Rome Lab-funded work, or otherwise related to Rome Lab-funded work.

P. Pazandak, J. Srivastava, Experience with the DAMSEL Multimedia Specification Language. Proceedings of the Society for Photonics and Electronics (SPIE) East, Conference on Multimedia, Philadelphia, September, 1995.

P. Pazandak, J. Srivastava, The Design of the DAMSEL Multimedia Specification Language. Proceedings of the Conference on Protocols for Multimedia Systems, Salzburg, Austria, October, 1995.

Y. Won, J. Srivastava, Scheduling Delivery of Video in a Distributed Multimedia Environment. Proceedings of the Conference on Protocols for Multimedia Systems, Salzburg, Austria, October, 1995.

J. Huang, Real-Time Scheduling Technology for Continuous Multimedia Applications, Lecture Notes of the 3rd ACM Multimedia Conference, November 1995.

D. Kenchammana-Hosekote and J. Srivastava, Storage and Retrieval of Compressed Audio and Video. Proceedings of the Society for Photonics and Electronics (SPIE) West, Conference on Multimedia, San Jose, January, 1996.

J. Huang, Y. Wang, and D. Kenchammana-Hosekote, Decentralized End-to-End Scheduling for Continuous Multimedia, Proceedings of the 6th International Workshop on Network and Operating System Support for Digital Audio and Video, Japan, April 1996.

J. Srivastava and D. Kenchammana-Hosekote. Design of an Integrated Toolkit for Multimedia Programming. Proceedings of the 6th IEEE Dual-Use Technologies and Applications, pp. 119-124, Utica, New York, June 1996.

J. Huang and P.-J. Wan, On Supporting Mission-Critical Multimedia Applications, Proceedings of the 3rd IEEE International Conference on Multimedia Computing and Systems, Hiroshima, Japan, June 1996.

P. Pazandak, J. Srivastava, The Temporal Component of DAMSEL. Proceedings of the 3rd IEEE International Conference on Multimedia Computing Systems, Hiroshima, Japan, June, 1996.

J. Huang and P. Allalaghatta, The Mercuri Multimedia Laboratory at Honeywell. IEEE Multimedia, Vol. 3., No. 2, Summer 1996.

P. Pazandak and J. Srivastava, Metrics for Evaluating ODBMS Functionality to Support MMDBMS. Proceedings of the Conference on Multi-Media Database Management Systems, Blue Mountain Lake, NY, August, 1996.

Y. Won, J. Srivastava, Tradeoffs in the Design of a Hierarchical Storage Based Video Server. Proceedings of the Conference on Multi-Media Database Management Systems, Blue Mountain Lake, NY, August, 1996

J. Huang, System Resource Management for Mission-Critical Multimedia Applications. Lecture Notes and Video Tape of Computer Science Colloquium, University-Industry Television for Education, October 1996.

D. Wijesekra, J. Srivastava, Quality of Service (QoS) Metrics for Continuous Media. Multimedia Tools and Applications Journal, 1996, Kluwer Academic Publishers. (Earlier version published in Proceedings of the Conference on Protocols for Multimedia Systems, Salzburg, Austria, October, 1995.)

J.-W. Chang, J. Srivastava, Spatial Match Retrieval Using Signature Files for Iconic Image Databases. Proceedings of the IEEE International Conference on Multimedia Computing Systems, Ottawa, Canada, June, 1997.

J. Huang, Y. Wang, N.R. Vaidyanathan, F. Cao, GRMS: A Global Resource Management System for Distributed QoS and Criticality Support. Proceedings of the 4th IEEE International Conference on Multimedia Computing and Systems, June 1997.

J. Huang, D. Kenchammana-Hosekote, M. Agrawal, and J. Richardson, P*resto* —A System Environment for Mission-Critical Multimedia Applications. Journal of Real-Time Systems, July 1997. (Earlier version appeared in Proceedings of the 6th IEEE Dual-Use Technologies and Applications, pp. 103-108, Utica, New York, June 1996.)

Y. Won and J. Srivastava. Distributed Service Paradigm for Remote Video Retrieval Request. Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing, Portland, Oregon, August 1997.

P.N. Pazandak and J. Srivastava, An Evaluation of Object-Oriented Database Management Systems to Meet the Needs of Multimedia Data Management. To appear in IEEE Multimedia Journal, Fall 1997.

D. Kenchammana-Hosekote and J. Srivastava, I/O Scheduling of Continuous Media Streams in a Video-On-Demand (VOD) Server. To appear in ACM-Springer/Verlag Multimedia Journal, 1997.

Y. Won, J. Srivastava, Scheduling of Video Delivery in a Distributed Environment, accepted for publication in Multimedia Tools and Applications Journal, Kluwer Academic Publishers.

Y. Won, J. Srivastava, Distributed Service Paradigm for Remote Video Retrieval Request. Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC), Portland, OR, August 1997.

W.J. Lee, S. Difu, D. Wijesekra, J. Srivastava, D. Kenchammana-Hosekote, Experimental Performance Evaluation of a Continuous Media File System (P*resto*), submitted to the ACM-ISCA Conference on Information & Knowledge Management (CIKM), 1997.

W.J. Lee, S. Difu, D. Wijesekra, R. Harinath, S. Wadhwa, J. Srivastava, D. Kenchammana-Hosekote, Experience in Building a Multimedia Programming Environment, submitted to ICPADS Conference, 1997.

9.2 Patents

Honeywell has submitted the following patent applications related to P*resto* and Sonata:

J. Huang, Criticality and Quality-of-Service (QoS) Based Resource Management. March 1997.

J. Huang and Y. Wang, Ripple Scheduling for End-to-End Global Resource Management. March 1997.

J. Richardson and M. Agrawal, Object State Change and History Management System, November, 1997.

Section 10
Software Documentation

10.1Application Development Tools

This section documents the design of the Program Development Tool (PDT). It illustrates the modularity of the program, and describes the classes and methods used. Although the design is fairly language independent, familiarity with object-oriented languages (preferably Java) or methodology is required.

The Program Development Tool allows the user to create a distributed multimedia application. The application program is then transferred to the User Interface Development Tool (UIDT) to specify the user-visible behavior of the components of the system. Finally, the program is transferred to the Sonata run-time, which executes it after some pre-processing.

The PDT runs under any machine that supports the Java Virtual Machine. It is written in Java, so it is portable to any platform that supports it.

The remainder of this section describes the classes and methods that are defined in the system. It describes the private and public APIs that the classes support. For more details on the methods and APIs, browsing through the source code is recommended. The source code is well commented and easy to read.

10.1.1 Block Class

The block class is used to represent the information stored for a particular block. Methods represent the operations that the class can handle.

10.1.1.1 Private Data

The private information stored in the block class is

- block_name: String
- block_num: int
- block_dimension: Rectangle
- port_list: Vector
- param_list: Vector
- selected: boolean
- foreign: boolean

10.1.1.2 Constructor

- Block ()
  This function creates and initializes the contents of the Block class. Usually invoked

whenever a new block object is created
Return Value: None

10.1.1.3 Public methods

The public methods for this class are

- set_block_name (String name)
  This function sets the block name of the Block object. Usually invoked to set the block name.
  Return Value: None
- get_block_name ()
  This function gets the block name of the current Block object. Usually invoked to retrieve the block name.
  Return Value: String block_name
- Similar methods exist for setting and getting the other data objects—block_num, block_dimension, selected and foreign.
- add_port (Port newport)
  This function adds a port to the existing list of ports.
  Return Value: None
- add_param (Param newparam)
  This function adds a parameter to the existing list of parameters.
  Return Value: None
- setParamValue (String p_name, String p_value)
  This function sets the parameter value of p_name to the specified value (p_value).
  Return Value: None
- print_param_name (PrintStream outp)
  This function prints the parameter information to the output stream. Usually this function is invoked when saving to a file is called.
  Return Value: None
- print_param_value (PrintStream outp)
  This function prints the parameter value to the output stream. Usually, this function is invoked when saving to a file is called.
  Return Value: None
- draw (Graphics gc)
  This function draws the block on the screen canvas. It is invoked whenever the refresh or update canvas function is called.
  Return Value: None
- move (int dx, int dy)
  This function moves the block on the screen by delta values in the X and Y directions
  Return Value: None

10.1.2 Connection Class

The connection class represents the list of connections for the current application. The methods in this class represent the actions that can be performed on these classes.

10.1.2.1 Private Data

The private information stored in the connection class is

- from_block_num: int
- from_port_name: String
- to_block_num: int
- to_port_name: String

10.1.2.2 Constructor

The default constructor is used for this class.

10.1.2.3 Public Methods

The public methods for this class are

- set_values (int fromBlockNum, String fromPortName, int toBlockNum, String toPortName)
  This function sets the values for the internal data elements.
  Return Value: None
- draw (Graphics gc, Block fromBlk, Block toBlk)
  The draw function draws the connection on the screen canvas. This connection is drawn from the fromBlk to the toBlk. The line coordinates and the arrow direction are calculated in this function.
  Return Value: None

10.1.3 ProgFile Class

This class represents the information for a particular application program. Each program is stored in a ProgFile object.

10.1.3.1 Private Data

The private information stored in the ProgFile class is

- BlockList: Vector
- ConnectionList: Vector
- max_block_num: int
- ClipBlockList: Vector
- ClipConnList: Vector
- CanvasWidth: int

- CanvasHeight: int
- CanvasLocation: String

10.1.3.2 Constructor

- ProgFile ()
  The constructor for this class creates space for Blocks and Connections. The data elements are initialized.
  Return Value: None

10.1.3.3 Public Methods

The public methods for this class are

- addBlock (String BlockName)
  addBlock (int BlockNum, String BlockName)
  addBlock (int BlockNum, String BlockName, Rectangle dim)
  This functions adds the block named BlockName to the list of blocks in the application program.
  Return Value: None
- print_block_list (PrintStream outp)
  print_block_list (Vector blklist, PrintStream outp)
  This function prints the block information of the application. This function is typically invoked when the save function is called.
- addConnection (int fromBlockNum, String fromPortName, int toBlockNum, String toPortName)
  This function adds the connection as specified by the parameters to the list of connections in the application program.
  Return Value: None
- print_connection_list (PrintStream outp)
  print_connection_list (Vector connlist, PrintStream outp)
  This function prints the connection information of the application. It is typically invoked whenever the save function is called.
  Return Value: None
- draw (Graphics gc)
  This function is typically called to draw the program information (blocks and connections) on the screen canvas. Updates on the screen are also done using this function.
  Return Value: None
- cutToClipBoard ()
  This function removes the selected blocks and puts them in the clipboard for future use. The clipboard is an offscreen buffer.
  Return Value: None
- copyToClipBoard ()
  This function only copies the selected blocks and puts them in the clipboard. It does not

remove them from the screen canvas.
Return Value: None

- pasteFromClipBoard ()
  This function may be used to copy the contents of the clipboard on to the screen canvas.
  Return Value: None

- is_blk_in_list (int block_num, Vector BList)
  This function lets you know if the block number block_num is in the list BList.
  Return Value: TRUE if yes, FALSE if not

- zoomIn ()
  This function zooms into a composite block. All the underlying blocks are exposed and a blue rectangle encloses the composite block.
  Return Value: None

10.1.4 SonataFile Class

The SonataFile class represents the information and procedures involved with the storage and retrieval of the application program data from the underlying file system.

10.1.4.1 Private Data

The private information stored in the SonataFile class is

- FileName: String
- DirName: String
- FileNameSet: boolean

10.1.4.2 Constructor

- SonataFile ()
  The constructor function for this class sets the private data fields to their default values.
  Return Value: None

10.1.4.3 Public Methods

The public methods for this class are

- setFileName (String dir, String file)
  This function sets the file name and directory name of the current file to the values specified as parameters.
  Return Value: None

- unsetFileName ()
  This function unsets the current file and directory name.
  Return Value: None

- getAbs ()
  This function returns the absolute file name of the current file, if set.
  Return Value: Absolute file name as a String

- saveToFile (ProgFile CurrProgFile)
  This function saves the current application program to the current file.
  Return Value: None
- openFile (ProgFile CurrProgFile)
  This function opens the current file and reads the program data from it. The application program data is stored in the parameter program file.
  Return Value: TRUE if successfully read, FALSE if not

10.1.5 SonataGUI Class

The SonataGUI class contains information pertaining to the graphical user interface and its representation.

10.1.5.1 Private Data

The private information stored in the SonataGUI class is

- CurrentFile: SonataFile
- CurrentProgFile: ProgFile
- mainCanvas: SonataCanvas
- isInit: boolean

10.1.5.2 Constructor

- SonataGUI ()
  The constructor function for this class defines the user interface hierarchy and sets up the Frame window. It creates the menu bar, the canvas and the push buttons needed for the application.
  Return Value: None

10.1.5.3 Public Methods

The public methods for this class are

- handleEvent (Event evt)
  This function handles user events, ranging from mouse clicks to menu action items. It triggers off the relevant module depending on the action. This is an override of the handleEvent method of the parent class.
  Return Value: TRUE if handled successfully, FALSE if not

10.1.6 SonataCanvas Class

The SonataCanvas class represents the information need to manage the canvas that appears on the screen. The handling of the user events over the canvas as well as the drawing of blocks and connections on screen are managed by this class.

10.1.6.1 Private Data

The private information stored in the SonataCanvas class is

- CanProgFile: ProgFile
- mode: int
- offScreen: Image
- imagewidth: int
- imageheight: int

10.1.6.2 Constructor

- SonataCanvas (ProgFile aProgFile)
  The constructor function for this class sets up the screen canvas with default values for background color, foreground color and mode. It also initializes the program application file to the one specified.
  Return Value: None

10.1.6.3 Public Methods

The public methods for this class are

- paint (Graphics gc)
  This function is called by the event handler to draw the blocks on the canvas. Usually, the refresh and initial draw commands make use of this function.
  Return Value: None
- mouseDown (Event evt, int x, int y)
  This function is called when ever the mouse is clicked over the canvas. A check is made to see what the users intentions are
  Return Value: TRUE if action has been handled, FALSE if not
- mouseDrag (Event evt, int x, int y)
  This function is invoked when ever the user drags the mouse over the canvas. A check is made to see what the user intends to do.
  Return Value: TRUE if the action has been handled, FALSE if not

10.1.7 Utility Classes

There are a number of other classes that are used in the system. Most of these are utility classes—they serve an important task in the system but are more general-purpose. This section describes these classes.

- BlockInfoDialog Class
  The function of this class is to bring up a dialog window that describes the properties of a block. The user can update these properties or just view them.
- BoldLabel Class
  This is an extension of the Label class and its primary aim is to provide labels in bold.
- ChangeParamDialog Class
  This class is used by the user to change the parameters of a particular block. It is usually invoked from the block information dialog window

- ConnectionInfoDialog Class
  This class displays the connection information during the creation of a new connection. It prompts the user to enter the to and from block numbers and port names.
- createBlockDialog Class
  This class is created whenever a Create New Block command is issued. The user is prompted for block information.
- EditSourceDialog Class
  This class is used to invoke a dialog window that allows the user to edit the current source code.
- Param Class
  This is a simple class to handle parameters.
- Port Class
  This is a simple class to handle ports.
- SelectFile Class
  This class is invoked to create a file selection dialog, from which the user can select a particular file.
- Sonata Class
  This class can be used to start the application.
- SonataButton Class
  This is an extension of the PushButton class.
- SonataText Class
  This class is an extension of the TextField class.
- WarningDialog Class
  This class allows the creation of warning dialog message windows.
- YesNoDialog Class
  This class is used to create a dialog window that prompts the user to choose Yes or No.

10.1.8 Conclusion

This section has documented the various classes used in the system design and the methods involved. It is a comprehensive list of functions that can be invoked from each of the classes. It is intended to serve as an API reference for future programmers of PDT.

It will hopefully serve as a guide to designing toolkits and graphical user interface applications. It is hoped that experiences with this system will help guide the way for future projects in this area.

10.2Continuous Media Server

10.2.1 Introduction

This section documents the design of the Continuous Media Server (CM Server). It illustrates the modularity of the program, and describes the functions, classes and methods used. The CM Server can be running as an independent application alone, so it allows the user to play the stored

MJPEG video files with a video functionality user interface regardless of the Active View System. A video clip as a field in AOC and WOC views can also invoke it.

10.2.2 Program Modules

The programs for CM Server are composed of two parts: server and client. There are some common modules used for both server and client and we have a text- version client (dummyClient) just for debugging. The detail source files are as follows:

10.2.2.1 Modules for Server

- main.c: CM server main file.
- network.c: file which includes network-related functions such Network Manager module, Proxy server module, and Video functions.
- cmStream.c: file which initializes CM stream-related data structures.
- UFSstream.c: file which includes UFS I/O Manager module.
- PFSstream.c: file which includes PFS I/O Manager module.
- ioUtil.c: file which includes some miscellaneous file I/O functions.

10.2.2.2 Modules for Client

- client.c: graphic-version CM client file including clientNetController, Tk/Tcl library interface functions, CM-Playback, and Parallax-control module.
- dummyClient.c: text-version CM client file.

10.2.2.3 Common Modules for Server/Client

- global.c: file which includes globally used functions.
- timeMeasurement.c: file which includes a function that gets current time.
- errorHandlers.c: file which includes error handling routines.

10.2.2.4 Orbix-related modules

- CM.idl: file which includes Orbix-version CM Server interfaces.
- CMS.cc: Orbix-made server-related file.
- CMC.cc: Orbix-made client-related file.

10.2.2.5 Executables

- server*: CM Server executable.
- client*: CM Client executable (graphic mode: Parallax version).
- dummyClient*: CM Client executable (text-mode).

10.2.3 Functions in Source Modules

10.2.3.1 Server Modules

10.2.3.1.1 main.c

- main(argc, argv): CM Server main function.

10.2.3.1.2 network.c

- networkManager(portNum): Network Manager main function which tells Orbix daemon that we have completed server's initialization.
- CM_Request_i::playMJPG(CM_User_ptr,name,fileSys,rate,clientPid,streamType): Sends an open (initial Play) request from client to server with initial parameters.
- CM_Request_i::ff(clientPid, newRate, env): Fast-Forward Video function.
- CM_Request_i::play(clientPid, env): Slow-Forward Video function.
- CM_Request_i::pause(clientPid, env): Pause Video function.
- CM_Request_i::resume(clientPid, env): Resume Video function.
- CM_Request_i::stop(clientPid, env): Stop Video funciton.
- ProxyServer(ptr_to_requestFromClient): Main module for Proxy Server which gets requests from clients, processes the request and sends the data to QoS Manager.

10.2.3.1.3 qosManager.c

- AdmController(rate): Admission Controller which checks disk I/O bandwidth to determine if we accept the new request.

10.2.3.1.4 cmStream.c

- CleanStream(index): Initializes CM stream data structures.

10.2.3.1.5 UFSstream.c

- UFSMjpegIOManager(index): UFS I/O Manager threshold.

10.2.3.1.6 PFSstream.c

- PFSMjpegIOManager(index): PFS I/O Manager threshold.

10.2.3.1.7 ioUtil.c

- FileSize(fileName): file I/O function to check file size.
- FileExist(fileName): file I/O function to check file existence.

10.2.3.2 Client Modules

10.2.3.2.1 client.c

- main(argc, argv): Main funciton of CM client.

- timeout(signal): Checks SIGALRM and lets the client quit when time out.
- sig_usr(sig_num): Interrupt handler for user-defined signal (signal handling function).
- clientNetController(argc, argv): Client Network Controller thread which sends input parameters to the relevant Proxy Server in CM Server and CM_Playback module to display MJPEG data on X-Window using Parallax libraries and card.
- myTclAppInit(Tcl_Interp): Initializes standard Tcl and Tk libraries, and loads a video function configuration file.
- CMPlayback(argc, argv): CM Playback function which initializes X-window & Parallax libraries, and displays MJPEG video frame.
- CM_User_i::putMJPGFrame(mjpg_frame, length, env): Function which reads MJPEG data from CM Server via Orbix.

10.2.3.2.2 dummyClient.c:

Functions are the same as those in client.c

10.2.3.3 Common Modules

10.2.3.3.1 global.c

- CleanUp(arg): Closes a Socket (only used in TCP/IP version).

10.2.3.3.2 timeMeasurement.c

- GetCurrentTime(): Retrieves the current system time.

10.2.3.3.3 errorHandlers.c

- errorQuit(msg, ...): Returns error message and exits.
- errorReturn(msg, ...): Returns error message (no exit).
- note(msg, ...): Dumps out some messages for notice (debugging).
- warning(msg, ...): Dumps out some messages for warning.

10.2.3.4 Orbix-related modules

- interface CM_User {
  putMJPGFrame(mjpg_frame, length);
  }
- interface CM_Request {
  playMJPG (to_where, name, filesys, rate, clientPid, streamType);
  ff (clientPid, newRate);
  sf (clientPid, newRate);
  play (clientPid);
  pause (clientPid);
  resume(clientPid);

```
  stop (clientPid);
}
```

☆U.S. GOVERNMENT PRINTING OFFICE: 2001-610-055-10110